# Spring Boot 3 API Mastery

*Write scalable, reactive, and secure APIs for microservices with Spring Boot 3 and Java 21*

**Vincenzo Racca**



www.bpbonline.com

First Edition 2025

Copyright © BPB Publications, India

ISBN: 978-93-65898-088

*All Rights Reserved*. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

## LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

**To View Complete**
**BPB Publications Catalogue**
**Scan the QR Code:**

www.bpbonline.com

## Dedicated to

*My parents **Pino** and **Nunzia Racca**, whose love, support, and guidance have shaped who I am today*

# About the Author

**Vincenzo Racca** is a Software Engineer specializing in Java, Spring, and Cloud technologies. He has enhanced his technical expertise by earning certifications in the Spring Framework, Kubernetes, and AWS. Vincenzo shares his knowledge by publishing articles on Java, Spring, and Cloud-native development.

# About the Reviewers

- **Lucas Fernandes** is passionate about software engineering, specializing in the development of highly scalable, resilient, and fault-tolerant applications. He has extensive expertise in all phases of the software development lifecycle, from conception to deployment. A strong advocate for microservices and event-driven architectures, Lucas has significant experience with technologies such as Java, Kotlin, the Spring ecosystem, Kafka, Kubernetes, and AWS services. As a cloud computing specialist, he has deep expertise in public cloud platforms, particularly AWS, where he holds two certifications.

  Currently, he works at Iteris Consulting and Software as a Tech Lead and Solutions Architect, designing and developing critical applications, especially in the financial sector. Recently, he also embarked on a new journey as a writer on Medium, where he shares his knowledge and experiences in the tech industry, promotes best practices, and inspires the developer community.

- **Modassir Kashani** is a skilled Software Engineer with extensive expertise in the Fintech domain. He has a wealth of experience in API development, microservices architecture, cloud deployment, and site reliability engineering. Modassir specializes in designing and developing payment applications, with a focus on crafting innovative solutions to real-world challenges using payment APIs.

  Currently, he is pursuing a Master's degree in Artificial Intelligence at Queen Mary University of London.

# Acknowledgement

I would like to thank everyone who encouraged me to write this book. A special thanks to my girlfriend, Anna, my sister, Sonia, and my friend, Chiara, for their unwavering support.

I am deeply grateful to BPB Publications for believing in me and giving me the opportunity to write a book that holds such personal significance. Thank you to their entire team for helping me refine and enhance the content of this work.

I also extend my gratitude to my colleagues, both past and present, who have contributed to my growth and development by generously sharing their knowledge and experiences with me.

Finally, my heartfelt thanks go to all the readers of this book for placing their trust in me. Your support is my greatest motivation to continue sharing knowledge and learning together.

# Preface

In an ever-evolving technological landscape, the ability to design and implement scalable, secure, and efficient APIs has become a cornerstone of modern software development. APIs are not merely tools for communication between systems—they form the backbone of microservices architectures, enabling businesses to innovate and adapt to change swiftly.

This book was born out of the realization that many developers face challenges in creating APIs that not only meet immediate requirements but are also robust, future-proof, and aligned with industry best practices. It is designed to provide readers with a comprehensive guide to mastering API design and implementation using Spring Boot, one of the most powerful and versatile frameworks available today.

Throughout these chapters, we will explore how to build APIs using Spring Boot, covering synchronous APIs such as REST, GraphQL, and gRPC, as well as asynchronous, event-driven APIs powered by Apache Kafka and Spring Cloud Stream. We will delve into topics like observability, distributed tracing, and security using OAuth2 and OpenID Connect. Finally, we will deploy microservices on Docker and Kubernetes, completing the journey from design to production.

This book is intended for developers and architects seeking to enhance their understanding of API design using Spring Boot in the context of cloud-native applications. It is a practical guide filled with real-world examples to help you not just learn but apply these concepts to your projects.

I hope this book serves as a valuable resource in your journey as a developer. Thank you for embarking on this learning experience with me.

**Chapter 1: Introduction to REST Architecture and API-first Approach -** This chapter covers a journey into microservice development. It navigates the intricacies of synchronous vs. asynchronous communication, elucidates the foundational principles of REST architecture, and delves into the nuances of managing API versioning. The chapter culminates with an exploration of the API-first approach, illuminating its significance in crafting scalable and well-designed microservices. Through a blend of theoretical insights and practical considerations, the reader gains a solid foundation for embarking on the subsequent chapters, setting the stage for a comprehensive exploration of API development in the evolving landscape of microservices architecture.

**Chapter 2: Reactive REST APIs with Spring WebFlux -** This chapter covers the reactive programming paradigm and explains how it differs from traditional approaches. It introduces Spring WebFlux, a powerful framework for building high-performance, non-blocking APIs. You will explore concepts like reactive streams and event loops while creating scalable REST APIs. Practical examples illustrate how reactive programming enhances responsiveness, making applications more efficient in handling concurrent requests.

**Chapter 3: Easily Scalable APIs with Virtual Threads -** This chapter covers the power of Virtual Threads introduced in Java 21 for building highly scalable synchronous APIs. It explains how Virtual Threads revolutionizes concurrency management, making it possible to handle numerous requests efficiently without blocking. The reader will learn to integrate Virtual Threads with Spring MVC, analyze performance benefits, and explore real-world examples of creating scalable, production-ready APIs.

**Chapter 4: GraphQL with Spring Boot –** This chapter covers the fundamentals of GraphQL, an alternative to REST for building flexible and client-focused APIs. It covers GraphQL schema, queries, and mutations, with practical examples using Spring Boot. You will understand how GraphQL enables efficient data fetching, reduces over-fetching and under-fetching, and supports evolving API requirements. By the end, the reader will be equipped to design and implement GraphQL APIs tailored to complex client needs.

**Chapter 5: Designing APIs with gRPC -** This chapter covers gRPC, a high-performance, language-neutral framework for communication between microservices. It explores the use cases, benefits, and implementation of gRPC in a Spring Boot application. The reader will learn how Protocol Buffers work for defining services and messages, and compare gRPC with REST. Practical examples demonstrate how to create scalable, low-latency APIs, making it an essential tool for high-throughput distributed systems.

**Chapter 6: Asynchronous APIs with Spring Cloud Stream and Apache Kafka -** This chapter covers building event-driven APIs for asynchronous communication between microservices. It explains how to use Apache Kafka as a message broker and Spring Cloud Stream for abstracting message processing. The reader will explore patterns like pub/sub and message partitioning. Through hands-on examples, the reader will understand how to implement APIs that handle asynchronous events efficiently.

**Chapter 7: Centralized Security with Spring Cloud Gateway -** This chapter covers how to implement centralized authentication and authorization using Spring Cloud Gateway. It explores OAuth2 and OpenID Connect protocols, integrates Keycloak as an Identity Provider, and configures microservices as OAuth2 clients and resource servers.

**Chapter 8: Observability and Monitoring -** This chapter covers observability and its importance in maintaining reliable APIs. It covers logging, metrics, and distributed tracing using tools like Grafana, Prometheus, and Tempo. The reader will learn to aggregate logs, monitor performance metrics, and trace request flows across microservices. By implementing observability best practices, the reader can quickly diagnose issues, optimize performance, and ensure a seamless user experience.

**Chapter 9: Deploying Applications on Kubernetes with Kind -** This chapter covers the deployment of microservices using Docker and Kubernetes. It focuses on Kind, a lightweight tool for running Kubernetes clusters locally. The reader will explore concepts like Pods, Deployments, and Services, and understand how to manage scaling and networking. Hands-on examples guide the reader through deploying APIs, ensuring they are production-ready and aligned with cloud-native principles.

# Code Bundle and Coloured Images

Please follow the link to download the
***Code Bundle*** and the ***Coloured Images*** of the book:

## https://rebrand.ly/9db530

The code bundle for the book is also hosted on GitHub at
**https://github.com/bpbpublications/Spring-Boot-3-API-Mastery**.
In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

# Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

**errata@bpbonline.com**

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

**business@bpbonline.com** for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

## Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

## If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

**https://discord.bpbonline.com**



# Table of Contents

**1. Introduction to REST Architecture and API-first Approach ..... 1**

Introduction ..... 1

Structure ..... 1

Objectives ..... 2

Introduction to microservice architecture ..... 2

Synchronous Vs. asynchronous communication ..... 3

*Synchronous communication ..... 4*

*Asynchronous communication ..... 5*

Fundamentals of REST architecture ..... 7

Restful API design ..... 8

*Richardson Maturity Model level 0: The swamp of POX ..... 9*

*Richardson Maturity Model level 1: Resources ..... 10*

*Richardson Maturity Model level 2: HTTP verbs ..... 10*

*Richardson Maturity Model level 3: HTTP verbs ..... 12*

*The concept of API idempotence ..... 12*

*API versioning strategies ..... 14*

*Final considerations on REST API design ..... 15*

OpenAPI specification and tools ..... 16

API-first approach ..... 19

Conclusion ..... 19

Points to remember ..... 19

**2. Reactive REST APIs with Spring WebFlux ..... 21**

Introduction ..... 21

Structure ..... 22

Objectives ..... 22

Introduction to reactive paradigm ..... 22

Core concepts of the reactive framework ..... 24

Event Loop vs. Thread-per-request ..... 30

REST APIs with Spring WebFlux ..... 31
*Installation of Java 21* ..... *32*
*Installation of Docker Desktop* ..... *32*
*Installation of HTTPie* ..... *33*
*Set up the Spring WebFlux project* ..... *33*
*Designing the product domain object* ..... *35*
*Getting started with Spring WebFlux* ..... *36*
*Exception handling in Spring WebFlux* ..... *54*
*Logging request and response in Spring WebFlux* ..... *58*
*Functional endpoints in Spring WebFlux* ..... *60*
Reactive REST client with Spring WebFlux ..... 61
Testing reactive APIs ..... 63
Spring MVC vs. Spring WebFlux performance ..... 66
Conclusion ..... 69
Points to remember ..... 69
Exercises ..... 69

**3. Easily Scalable APIs with Virtual Threads ..... 71**
Introduction ..... 71
Structure ..... 71
Objectives ..... 72
Understanding virtual threads in Java ..... 72
*Platform threads vs. virtual threads* ..... *72*
*Virtual threads to simplify scalability in Java* ..... *73*
*Virtual threads in detail* ..... *73*
Getting started with Spring MVC with virtual threads ..... 77
*Introduction to Spring MVC* ..... *77*
*Spring MVC in practice* ..... *79*
*Paginated API in Spring MVC* ..... *90*
A new modern synchronous REST client: RestClient ..... 99
Testing in Spring MVC ..... 104
Spring MVC performance comparison: Platform vs. virtual threads ..... 106
Choose between virtual threads and WebFlux ..... 108

Conclusion ..... 108
Points to remember ..... 109
Exercises ..... 109

**4. GraphQL with Spring Boot ..... 111**
Introduction ..... 111
Structure ..... 112
Objectives ..... 112
Fundamentals of GraphQL ..... 112
*Schemas and types ..... 113*
*Queries and mutations ..... 114*
Implementing GraphQL APIs with Spring Boot ..... 118
*GraphQL N + 1 problem ..... 128*
*Error handling in Spring for GraphQL ..... 132*
*Optimizing GraphQL with document parsing caching ..... 134*
*GraphQL client in Spring ..... 137*
Testing GraphQL APIs ..... 138
Comparing REST and GraphQL ..... 140
Conclusion ..... 141
Points to remember ..... 142
Exercises ..... 142

**5. Designing APIs with gRPC ..... 143**
Introduction ..... 143
Structure ..... 144
Objectives ..... 144
Overview of gRPC ..... 144
*The Protocol Buffers ..... 145*
*Communication in gRPC ..... 147*
*Comparison of gRPC and REST ..... 148*
Integrating gRPC into Spring Boot applications ..... 150
*Implementation of the gRPC server ..... 151*
*Implementation of the gRPC client ..... 157*

Testing gRPC APIs ... 162
Conclusion ... 166
Points to remember ... 167
Exercises ... 167

**6. Asynchronous APIs with Spring Cloud Stream and Apache Kafka ... 169**
Introduction ... 169
Structure ... 170
Objectives ... 170
Asynchronous APIs ... 171
*HTTP polling ... 171*
*Webhook ... 172*
*HTTP Streaming ... 173*
*WebSocket ... 175*
*Publish/Subscribe ... 176*
Introduction to Apache Kafka ... 177
Introduction to Spring for Apache Kafka ... 179
The Spring Cloud Stream project ... 181
*An overview of Spring Cloud Function ... 182*
*Main concepts in Spring Cloud Stream ... 184*
Message-driven microservices with Spring Cloud Stream ... 186
*Getting started with Spring Cloud Stream ... 190*
*Implementing a producer with Spring Cloud Stream ... 190*
*Implementing a consumer with Spring Cloud Stream ... 196*
*Spring Cloud Stream at work ... 201*
The AsyncAPI specification ... 203
Error handling in Spring Cloud Stream ... 206
Testing Spring Cloud Stream applications ... 209
Conclusion ... 213
Points to remember ... 214
Exercises ... 214

**7. Centralized Security with Spring Cloud Gateway ..... 215**
Introduction ..... 215
Structure ..... 216
Objectives ..... 216
Introduction to OAuth2 and OpenID Connect ..... 216
*JSON Web Token ..... 217*
*OpenID Connect protocol ..... 218*
*OAuth2 protocol ..... 220*
Using Keycloak as an identity server ..... 222
*Running Keycloak locally ..... 222*
*Configuring Keycloak ..... 223*
API Gateway with Spring Cloud Gateway ..... 224
Getting started with Spring Cloud Gateway ..... 226
*Resilience with Spring Cloud Gateway ..... 228*
*Implementing retry with retry filter ..... 228*
*Implementing circuit breaker with Resilience4J ..... 229*
OAuth2 client and resource servers with Spring Security ..... 233
*Configuring the OAuth2 client on Keycloak ..... 234*
Configuring the OAuth2 client on Spring Security ..... 235
Spring Session with Redis to manage sessions ..... 241
Configuring the resource server on Spring Security ..... 242
Access token management in the OAuth2 client ..... 243
Resource server implementation ..... 244
Role based access control with Spring Security ..... 246
Roles claim in the token ID ..... 246
Protecting APIs with RBAC in the resource server ..... 248
Testing OIDC and OAuth2 flow with Spring Security ..... 251
Configuring OAuth2Client for SPA ..... 253
Conclusion ..... 255
Points to remember ..... 256
Exercises ..... 256

**8. Observability and Monitoring ..... 257**
Introduction ..... 257
Structure ..... 258
Objectives ..... 259
Log aggregation with Grafana Loki ..... 259
*Logging in Spring Boot ..... 259*
*Working with Loki ..... 260*
Metrics with Spring Boot Actuator and Prometheus ..... 262
Tracing with Micrometer, OpenTelemetry, and Grafana Tempo ..... 267
Monitoring with Grafana ..... 270
Conclusion ..... 276
Points to remember ..... 276
Exercises ..... 276

**9. Deploying Applications on Kubernetes with Kind ..... 277**
Introduction ..... 277
Structure ..... 277
Objectives ..... 278
Spring Boot with Docker ..... 278
*Introduction to container technology and Docker ..... 278*
*Containerizing a Spring Boot application ..... 281*
*Running Spring Boot containers with Docker Compose ..... 284*
Introduction to Kubernetes ..... 287
*Deployment and service in Kubernetes ..... 289*
Deploy your API locally with Kind ..... 292
*Installation and creation of the cluster ..... 292*
*Deploy Easyshop to Kubernetes ..... 294*
*Writing deployment and service ..... 295*
*Running the cluster ..... 302*
Conclusion ..... 305
Points to remember ..... 306
Exercises ..... 306

**Index**

# Chapter 1
# Introduction to REST Architecture and API-first Approach

## Introduction

In today's rapidly evolving landscape of software development, the advent of microservices architecture has revolutionized how applications are designed, built, and deployed. At the heart of this architectural paradigm lies the pivotal role of **application programming interfaces** (**APIs**), serving as the cornerstone for communication and interaction between microservices. In this chapter, we embark on a journey to explore the fundamental principles and practices that underpin the development of robust and scalable APIs within a microservices environment. Our exploration begins with a fundamental distinction between synchronous and asynchronous communication paradigms, illuminating the strengths and trade-offs of each approach in facilitating seamless interaction between microservices. We will delve into the **Representational State Transfer** (**REST**) architecture, unraveling its core principles of statelessness, uniform interface, and resource-based design. In addition, we will discuss the advantages of using the API-first approach with the OpenAPI specification.

## Structure

In this chapter, we will discuss the following topics:

- Introduction to microservice architecture
- Synchronous Vs. asynchronous communication

- Fundamentals of REST architecture
- Restful API design
- OpenAPI specification and tools
- API-first approach

# Objectives

By the end of this chapter, you will have a clear understanding of synchronous and asynchronous communication between microservices, enabling you to evaluate and select the most appropriate method to use based on your use cases. You will appreciate REST architecture in detail, knowing best practices so that you can write clear APIs for your clients through the OpenAPI specification.

# Introduction to microservice architecture

In recent years, microservices architecture has become increasingly popular in enterprise projects. In previous years, projects were mostly written with a monolithic architecture. In this architecture, the entire application is developed, tested, deployed, and scaled as a single unit.

*Figure 1.1* shows a typical monolithic architecture where an entire team works on the same source code. The following application shows the reads and writes from a single database:

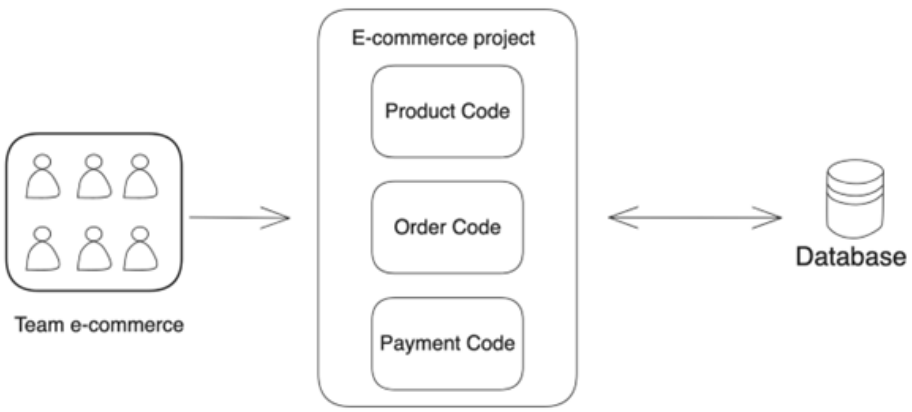



***Figure 1.1:*** *Example of monolithic architecture*

Microservice architecture divides an application into a set of independent and modular services which is responsible for a specific functionality. Each service in a microservice architecture can be developed, deployed, and scaled independently. Services communicate with each other through well-defined APIs and often use lightweight protocols such as HTTP or asynchronous messaging.

*Figure 1.2* shows a typical microservice architecture, in which each (micro)team works on a service to facilitate parallelization of developments. Each microservice reads from its own database, refer to the following figure for clarity:

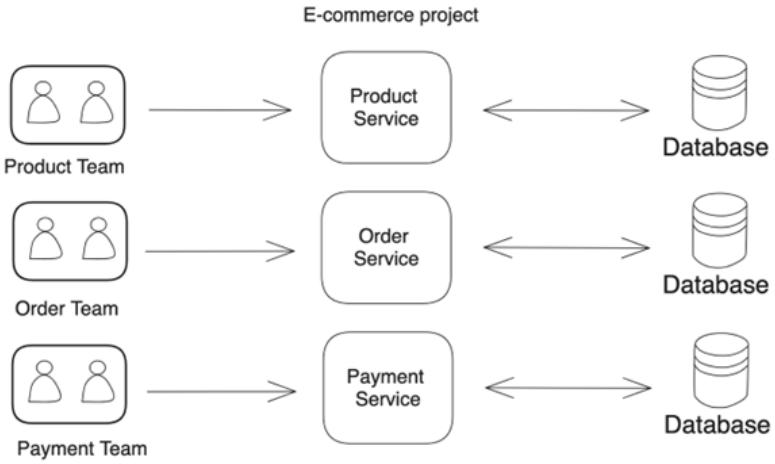



***Figure 1.2**: Example of microservice architecture*

**Note: In this architecture, each microservice reads from its own dedicated database, which reduces dependencies and promotes scalability and fault isolation. Sharing a database across multiple services can lead to tightly coupled services, increased risk of data conflicts, and difficulties in scaling or deploying services independently. In certain cases, however, multiple services may share a database, typically when the data is small, low-frequency, or the services are closely related.**

So, which of the two architectures is better? There is no absolute answer; it depends on the project. If you need to develop a simple, small application that is unlikely to change over time, the monolithic architecture might be a good choice. If the application you need to develop is complex, then it may change over time. Microservices architecture is suitable for taking advantage of all the benefits of the cloud.

# Synchronous Vs. asynchronous communication

In the previous section, we explained that in a microservice architecture, the components are independent services that work with their codebase, and their own artifact is distributed independently. But how do the services communicate with each other?

The answer is through API. They are a set of rules that allow different software applications to communicate with each other. These rules define the interfaces and data formats used for sending and receiving requests and responses between different components of the system. We can define an API as a contract between the server and its clients.

In the context of microservices, APIs are critical because they allow services to communicate in a standardized way that is independent of each service's internal implementation. This promotes modularity, independence, and reusability of individual services within the microservice architecture. We can divide them into two macro groups, synchronous APIs and asynchronous APIs. The choice of these two types of APIs determines the type of communication between services, synchronous or asynchronous, precisely.

# Synchronous communication

In synchronous communication, the calling service (client) waits for an immediate response from the called service (server) before proceeding with the next execution. This type of communication is often based on protocols such as **Hypertext Transfer Protocol** (**HTTP**), where a request is sent, and the caller waits for a response before continuing. The most common type of API in microservices architecture, this type of communication comes under REST. We will discuss this in-depth in the next section. However, other types of APIs, such as **Graph Query Language** (**GraphQL**) and **Remote Procedure Calls** (**gRPC**), have taken hold in recent years. In brief, the client sends an HTTP request, waits, the server receives the request, processes it, and finally responds to the client.

An example of synchronous communication between two microservices, where the client, order service, makes an HTTP request and the server, product service, responds to it with an HTTP response is shown in *Figure 1.3* shows:

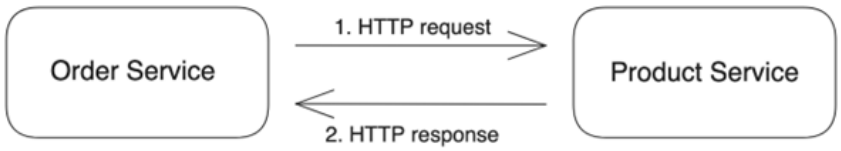



***Figure 1.3**: Synchronous communication*

Synchronous communication is suitable for operations where an immediate response is needed, and the caller depends on the result of the operation to proceed. However, it can be more vulnerable to scalability and availability problems because the caller is blocked while waiting for a response, and problems can occur if the server service becomes slow or unresponsive. Therefore, it is important to design applications to effectively handle these scenarios and ensure reliable synchronous communication between microservices, using patterns such as Retry, Circuit Breaker, and Service Mesh. For example, in an e-commerce application, synchronous communication might be used for calls to the search service (such as GraphQL queries) or payment gateway APIs, typically handled via an API gateway. Patterns like Retry and Circuit Breaker help manage transient failures by automatically retrying failed requests or gracefully handling downtimes, while Service Mesh provides advanced control over inter-service communication, including retries, load balancing, and traffic management. These patterns are especially useful in cloud deployments where robust and resilient communication between microservices is essential.

# Asynchronous communication

Asynchronous communication between microservices occurs when one service sends a message or request to another without waiting for the latter's response. This type of communication is very useful in scenarios where the caller does not need to be blocked while the receiving service processes the operation. We usually refer to this type of communication as event-based or event-driven architecture model.

In this model, a service called a publisher or producer sends events/messages to a communication channel, such as a queue or topic. One or more services, called subscribers or consumers, read the messages from the communication channel and process them.

With this type of communication, it reminds you of the first advantage which is the decoupling of producer and consumer services. The fact that there is a communication channel, such as a message broker (e.g., Kafka or RabbitMQ) or an event bus, that acts as *middleware*, allows us to handle any outages or unavailability of consumer services. Even if the consumer is down, the producer can still write the message to the channel. When it is up again, it can continue reading the messages.

*Figure 1.4* shows the order service that writes a message on a topic that is read by the consumers payment service and notification service:

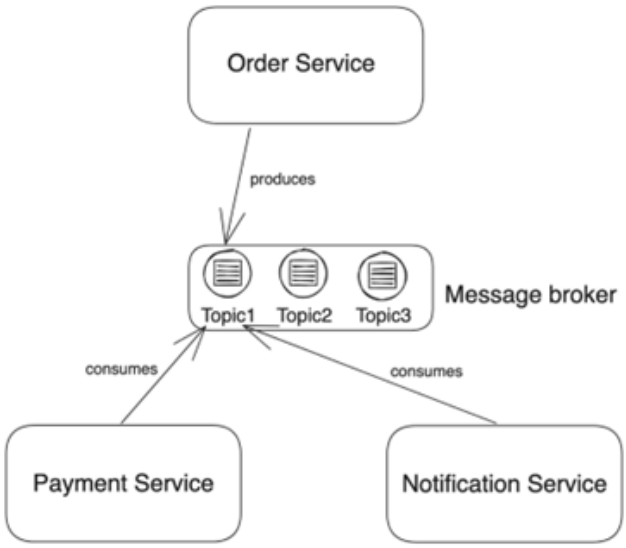



***Figure 1.4**: Asynchronous communication between two microservices*

Another advantage of this approach is that horizontal scaling is easier to actualize as it is sufficient to have multiple instances of the same service to scale message processing. *Figure 1.5* shows how instances of the same service share message reading. Each instance automatically reads from a different partition that contains multiple messages from other

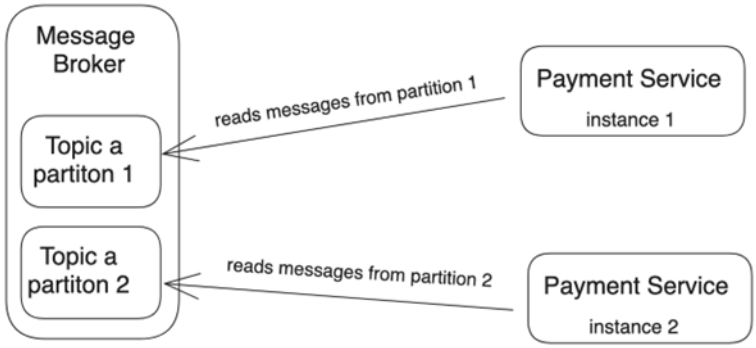

Message Broker
Topic a partiton 1
Topic a partiton 2
reads messages from partition 1
reads messages from partition 2
Payment Service instance 1
Payment Service instance 2

***Figure 1.5**: Two instances of the same service read from two different partitions of the same topic*

In synchronous communication, such as for REST services, it is not enough to have multiple instances of the same service. You have to use a tool that takes over the requests, checks which instance is available and then routes it to the actual service. This tool then needs to be configured according to the microservice. For example, one must define *what are the conditions for an instance of service X to be ready to receive traffic, such as configuring health and liveness probe.* Such tools can be simple Load Balancers, Reverse Proxies, or more complex API gateways. Order service figures out which of the instances are ready to receive traffic. If all the instances are ready, the Load Balancer, through its algorithm, such as a simple round-robin, chooses which instance will serve the incoming request. In the given diagram (*Figure 1.6*) load balancer checks the availability of order service instances based on configured readiness probe, and then routes the requests to available instances using its algorithms such as round-robin:

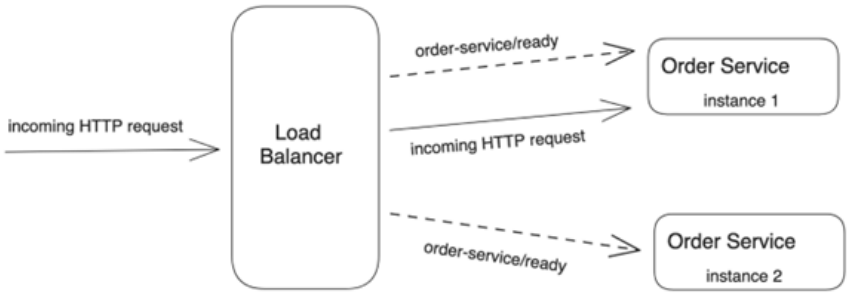



***Figure 1.6**: The load balancer chooses which instance to redirect traffic to based on the ready endpoint*

The type of asynchronous communication that we have explained, implements the pattern most used in event-driven architecture, namely the publish/subscribe pattern. However, there are other patterns, for more specific use cases, such as the event sourcing pattern, which keeps track, via events, of every change in business entities. The saga pattern, for example, manages distributed transactions between different services by breaking them into a series of steps, where each step depends on the outcome of the previous one. For

instance, in a flight booking system, the reservation or seat confirmation depends on the successful completion of the payment transaction.

Often, it is the synchronous communication that triggers the asynchronous flow. The order service makes an HTTP call to the payment service, which immediately responds to it with a status code 202 accepted. This status code in response indicates that the request has been taken care of but has not yet been processed. The actual business logic starts in asynchronous mode after the HTTP call. Once the processing of the business logic is finished, the payment service sends an event, which is read by the order service. The communication between order service and payment service occurs in two steps: the first is synchronous, and the second is asychronous. Have a look at *Figure 1.7* for a better understanding:

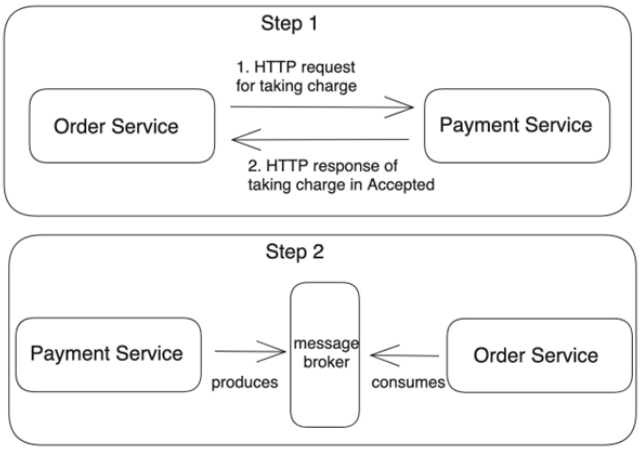



***Figure 1.7**: The communication between order service and payment service occurring in two steps*

# Fundamentals of REST architecture

In the previous section, we mentioned that in synchronous communication between microservices, the use of REST APIs is the most popular standard. The REST architecture is a model for designing distributed systems and building web services. Conceived by *Roy Fielding* in his 2000 doctoral dissertation, REST has become one of the main paradigms for web API design. As mentioned earlier, it is an architectural style, not a protocol unlike **Simple Object Access Protocol** (**SOAP)**, a protocol that was used before REST was developed. It gives guidelines on how to develop APIs, without specifying a particular protocol, although HTTP is almost always used for client/server communication, and the **JavaScript Object Notation** (**JSON**) format is used for data exchange. REST is based on a set of fundamental principles that promote scalability, reliability, and simplicity in the design of distributed systems. These principles guide the creation of web services that are easily understood, interoperable, and well-adapted to the web environment. The fundamental principles of REST are mentioned below:

- **Stateless**: The REST APIs must be stateless, which means that each request between client and server must contain all the information necessary to process it. The server must not maintain any information about the client's session state between requests. This principle simplifies server management and promotes horizontal scalability.
- **Uniform interfaces:** It must define a well-defined set of operations for accessing and manipulating resources. A resource is described using a **Uniform Resource Identifier** (**URI**). Manipulations of resources are performed by standard operations, such as retrieving a resource using the HTTP **GET** verb or creating a new resource using the HTTP **POST** verb.
- **Client-server architecture:** The client manipulates resources via URI, not having to know what is going on behind the scenes in the servers. For example, the client might request the deletion of a resource via its URI and the HTTP verb DELETE. The server, on the other hand, will delete the resource. Whether it is a physical deletion or a logical deletion is hidden from the client. The important thing is that, if the client tries to retrieve the resource again, it receives a 404 Not Found.
- **Cacheability**: REST promotes both client-side and server-side caching to enhance performance and reduce server load. Client-side caching allows the client to store responses locally, minimizing repeated requests, while server-side caching offloads work from the server by storing responses directly on the server or in middleware such as proxies and **Content Delivery Networks (CDNs)**. To guide caching behavior, server responses should include relevant cache headers, such as Cache-Control, indicating whether and how a resource can be cached by clients or intermediaries.
- **Layered system**: REST favors the use of a layered system. For example, an API REST could be *screened* by reverse proxy, load balances, and API gateway, to favor scalability and untie the responsibility of cross-cutting concerns (concepts that do not concern business logic, such as safety and logging). However, the division into layers should not impact the client.

Now that we have defined the fundamentals of REST, we can focus on best practices on how to write REST APIs.

# Restful API design

In order to write good REST APIs, you need to know the Richardson Maturity Model. This model gives guidelines for creating REST services, it consists of four levels (zero to three), each level getting closer and closer to what the ideal REST API should look like. If the constraints of level three are met, the so-called *Glory of REST* is achieved, and the API can be called RESTful. Let us look at all four levels in detail. We will show you an example of an API where the client requests the information of a user with ID one. *Figure 1.8* shows how the API evolves for each level:

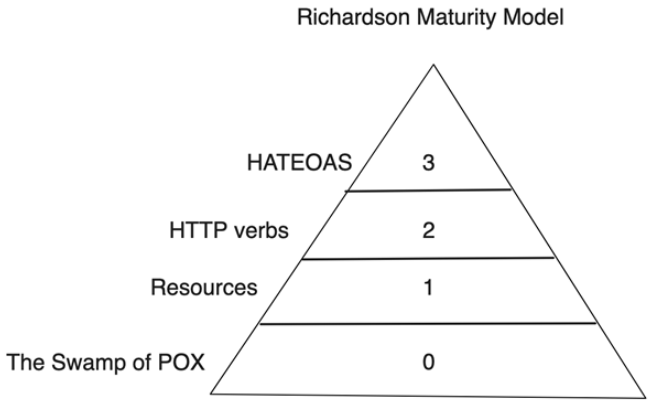



***Figure 1.8**: The pyramid of the Richardson Maturity Model*

# Richardson Maturity Model level 0: The swamp of POX

The concepts of resource and HTTP verbs do not exist. Services have a single URI and use a single HTTP method (usually POST). The HTTP is used only as a transport protocol. This mechanism is used, for example, in **Plain Old XML** (**POX**) applications and SOAP services. So for this level, the example will have the format of the request and response bodies in XML and not JSON.

The client request will have a URL like **http://localhost:8080/api** with HTTP POST verb, and a request body like the following:

```
<UserRequest>
    <id>1</id>
</UserRequest>
```

If the resource exists, the API will respond with the status code 200 OK and a response body like the following:

```
<User>
    <id>1</id>
    <name>Vincent</name>
    <surname>Smith</surname>
</User>
```

If the resource does not exist, the API will still respond with the status code 200 OK, and a body like the following:

```
<UserErrorResponse>
    <errorCode>1</errorCode>
    <reason>User not found</reason>
</UserErrorResponse>
```

# Richardson Maturity Model level 1: Resources

At this level, there is the concept of a resource. Resources can be any identifiable entity, such as data, or business objects. Each resource is identified with a specific URI. The concept of HTTP verbs does not exist at this level. The request will have a URL like **http://localhost:8080/api/users/1** with an HTTP POST verb and no request body.

If the resource exists, the API will respond with the status code 200 OK and a response body like the following:

```
{
    "id": 1,
    "name": "Vincent",
    "surname": "Smith"
}
```

If it does not exist, the API will still respond with the status code 200 OK, and a body like this:

```
{
    "errorCode": 1,
    "reason": "User not found"
}
```

# Richardson Maturity Model level 2: HTTP verbs

Here, the concept of HTTP verb (also called HTTP methods) enters, which can be associated with **Create**, **Read**, **Update**, **Delete (CRUD)** operations on resources. It can be summarized as the following:

- The HTTP GET verb requests to retrieve a single resource. This method is associated with the CRUD operation of read. The response will have a status code 200 OK if the resource exists, and 404 Not Found if it does not exist.
- The HTTP POST verb request to create a new resource. This method is associated with the CRUD operation of create. If the resource has been successfully created, the server responds with the status code 201 Created and inserts the response header (the Location key) valued with the URI of the newly created resource.

- The HTTP PUT verb requests to overwrite a resource. This method is the counterpart of the CRUD UPSERT (create and update) operation. If the resource exists, it is overwritten with the request body, and the server responds with status code 200 OK and a response body representing the newly overwritten resource. If the resource does not exist, a new resource is created, and the same response as a POST is returned (HTTP status code 201 and the Location key in the header).
- The HTTP DELETE verb request to delete a resource. This method is associated with the CRUD operation of the same name. If the resource exists and the delete operation is successful, usually the server responds with the status code 204 No Content, if it does not exist, it responds with a status code 404 Not Found.

As you may have noticed, in addition to using relevant HTTP methods for each type of operation, different status codes are also used depending on the type of response. The client, by analyzing the HTTP status alone, can figure out how the request went.

Returning to the example discussed so far, the request will have the same URL as the previous level, **http://localhost:8080/api/users/1**, this time, however, the HTTP verb used will be GET.

The response from the server, if the resource exists, will be the same as in the previous level. We will enrich the example by adding to the user resource as a new field called cars, which will be useful for the next level. This new field indicates any cars in the user's possession.

The response will have the status code 200 and the body look like the following code:

```
{
  "id": "1",
  "name": "Vincent",
  "surname": "Smith",
  "cars": [
     {
        "id":1,
        "plate":"BG929RF"
     },
     {
        "id":2,
        "plate":"AG929RF"
     }
  ]
}
```

# Richardson Maturity Model level 3: HTTP verbs

The final layer introduces the concept of **Hypermedia As The Engine Of the Application State** (**HATEOAS)**. It emphasizes the importance of including hypermedia links in REST API responses to guide the client through interactions with resources. The fundamental idea of HATEOAS is to make the server a kind of *map* to clients so that they have all the information they need to dynamically navigate through various resources and actions available.

Each resource contains links to all related resources and more, this allows the client to be told what other resources are linked to the resource, without burdening the body of the response, since these are simple links. An additional advantage is that the client can explore the calls made available by the server without any real documentation of the API (although tools such as OpenAPI and Swagger exist to document REST APIs). The resource can easily be enriched with new relationships to other resources by simply inserting new links.

Returning to the example discussed so far, the client's request remains the same as in the previous layer. The server response, if the resource exists, will have the status code 200 and the body similar to the following code:

```
{
     "id": 1,
     "name": "Vincent",
     "surname": "Smith",
     "_links": {
         "cars": {
             "href": "http://localhost:8080/api/users/1/cars"
         },
         "self": {
             "href": "http://localhost:8080/api/users/1"
         }
     }
 }
```

You may notice that this time the cars field simply contains a link that returns the list of all resources of type car that belong to the user with ID one.

# The concept of API idempotence

With HTTP verb handling, it is also important to know the concepts of safe methods and idempotent methods.

A method is considered **safe** if it does not change the state of the resource. The GET method is considered safe because it performs read-only operations.

On the concept of idempotency, there is some confusion in the REST environment. You will often find a definition like, *A method is idempotent if, invoked multiple times, it always produces the same result*.

I find it more relevant in the mathematical domain. In the REST domain, I prefer to define the concept of idempotency as, *A method is idempotent if, invoked multiple times, it produces no side effects on the server*.

Let us look in detail at what the idempotent HTTP methods are:

- The **`GET`** method is safe, it does not change the state of the resource, so it is also idempotent.
- The **`POST`** method is not idempotent, as multiple invocations of the same request would have the side effect of creating more resources.
- The **`PUT`** method is idempotent, if invoked multiple times, it always modifies the resource in the same way, producing no side effects.
- The **`DELETE`** method is idempotent, if invoked the first time on an existing resource, it deletes it. If it is invoked multiple times with the same request, no server-side action will be taken, creating no side effects.

Idempotent methods are methods in which you can safely apply the Retry pattern.

If we had used the first definition of idempotency, we would have had some difficulty explaining why the **`PUT`** and **`DELETE`** methods are idempotent. The **`PUT`** method would respond with status code 201 if the resource does not exist, and for subsequent calls, it would respond with 200. The **`DELETE`** method would respond 204 the first time on an existing resource. It would respond with a 404 when the same request is given subsequent times as the resource would no longer exist. *Table 1.1* shows a summary of the concept of idempotency, plus other HTTP methods:

| HTTP method | Idempotence | Safe |
|---|---|---|
| GET | yes | yes |
| POST | no | no |
| PUT | yes | no |
| DELETE | yes | no |
| PATCH | no | no |
| HEAD | yes | yes |
| OPTIONS | yes | yes |

***Table 1.1**: Idempotence and safety on HTTP methods*

However, nothing prohibits creating a **`POST`** method API that is idempotent. This can be useful in scenarios where the API needs to create a resource that cannot be overwritten (so the use of the **`PUT`** method is ruled out), and you want to implement the Retry pattern to have a better chance of the operation succeeding. A practical example is the creation of a payment transaction. In that case, with a normal POST, if retries were made because of the server delays in response, we would run the risk of creating multiple payment transactions.

*Figure 1.9* shows an example of a POST request that contains in the header a field, **`idempotencyToken`**, that makes it idempotent. The server verifies that the token is already in its database, then returns 200 without creating the resource:

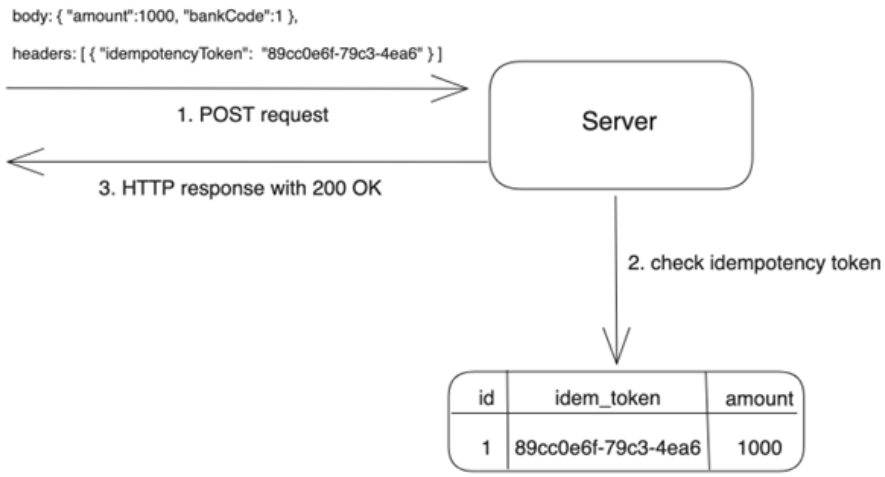



***Figure 1.9**: Example of a request/response of an idempotent POST API*

# API versioning strategies

Versioning is a fundamental practice in API design to manage their evolution and compatibility over time. When developing APIs, it is important to consider how future changes and additions will be handled to ensure that clients can continue to use them without interruption. There are several strategies for versioning APIs mentioned below:

- **Versioning by URI:** In this strategy, the version of the API is included directly in the URI of the request. For example, **`/api/v1/users`** and **`/api/v2/users`** represent two different versions of the same API. This approach is simple to implement but can make URIs more complex and less readable.
- **Versioning by headers:** In this strategy, the API version is specified via an HTTP header, such as Accept-Version: v1. This keeps resource URIs cleaner but requires management over the logic of HTTP headers by both the client and server.
- **Versioning by Content Negotiation:** It is a subset of the strategy in the previous point. In this strategy, the client specifies the

header, typically by defining a custom media type such as application/vnd.example.v1+json. This allows versioning at the content level rather than in the URI. However, it requires explicit content negotiation logic on both the client and server sides, and some clients might not fully support custom media types.

- **Versioning by query parameters:** In this strategy, the API version is specified as a query parameter in the URI of the request, for example, `/api/users?version=v1`.

Regardless of the strategy adopted, it is also important to provide a standard for version numbering. Very popular is the semantic version, which involves the use of version numbers with a specific format `<MAJOR>.<MINOR>.<PATCH>`. Refer to the explanations:

- **MAJOR**: It is increased when changes are made that are incompatible with previous versions or new incompatible features are introduced. For example, in the new version of the resource retrieval API, the structure of the response body will change.
- **MINOR**: It is increased when we add new features compatible with the previous versions. For example, in the new version of the search API, a new query parameter is added to filter on an additional field.
- **PATCH**: It increases when bugs are fixed, or minor changes are made that are compatible with the previous versions. For example, the new version of the API fixes a bug in the previous version where a field in the response body was being valued incorrectly.

There is no best strategy for API versioning, although versioning via URIs is the most common. However, it is important to choose the versioning mode before publishing the first version of the API so that clients are prepared for any future versions. In addition, all APIs in the application should follow the same strategy.

# Final considerations on REST API design

The question we can think about is that *in order to write a good REST API, does it necessarily have to meet all levels of the Richardson Maturity Mode*? The answer is no. Level three was designed when microservice architecture was not yet popular. We consider HATEOAS as a more useful feature for the classic monolith application. Going back to the previous example, in a microservice architecture it would be desirable for the user and car entities to be managed by two separate services. In that case, to implement HATEOAS, the user resource would have to contain links to another microservice, which perhaps is also located in a different domain. However, the microservice that manages users should not have the responsibility of providing clients with information regarding how to reach other microservices. One component that might have this responsibility is the service registry.

You can consider that you have written a good REST API if it meets the principles of levels one and two. In addition, we have listed some other tips below:

- Name of the resource within the URI mentioned as a plural (for example /users, not /user).
- The API defination should never contain verbs (for example, do not use URLs such as **`/users/getById`** or **`/users/save`**).
- Although REST does not constrain the data format, you should always prefer the JSON format.
- For the endpoint that retrieves all entities of a certain resource, provide paging, filtering, and sorting on the fields.
- Try not to return the resource to the client as it persists on the server side and do not let the client know too many details. To avoid this, you can use the **Data Transfer Object** (**DTO**) pattern.
- If the API is public, use an API versioning system, it prevents the creation of incompatibility problems for clients.
- Apply server-side rate limiting on APIs if necessary to prevent misuse and manage resource usage efficiently.
- Use consistent error codes and error messages to help clients understand issues easily and streamline troubleshooting.

# OpenAPI specification and tools

Earlier, we defined the API as a contract between the server and clients. But how do we formalize the API? The answer is through the OpenAPI specification. Formerly known as the swagger specification, it is a standard for describing APIs that allows it to be defined and documented in a standardized and interoperable way. It is a specification that provides a common framework for describing supported operations, required parameters, response formats, and other API-related information. In addition, it is a standard that can be interpreted by both humans and machines. Let us take a quick look at an example of an OpenAPI definition. The first section covers the version of the specification and information about the communication with the server.

```
openapi: 3.0.3
info:
  title: Sample API
  description: An optional description
  version: 0.1.9
servers:
  - url: 'http://api.example.com/v1'
    description: 'Main (production) server'
```

The second section covers the declaration and detail of each endpoint:

```
paths:
  /users:
    post:
      summary: Creates a new user.
      operationId: insertUser
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
      responses:
        '201':
          description: Created
          headers:
            Location:
              required: true
              schema:
                type: string
                format: uri
```

The final section covers the declaration of input / output components used by the endpoints (parameter and request / response bodies):

```
components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: integer
          example: 4
        name:
          type: string
          example: Arthur Dent
      required:
        - id
        - name
```

The OpenAPI document in the example is in **Yet Another Markup Language** (**YAML**) format, which is also the most widely used format in the OpenAPI standard. However, JSON format is also allowed. For more details on the OpenAPI specification, you can take a look at the following link: **https://swagger.io/specification/**.

Swagger is an ecosystem of tools that supports the use of OpenAPI. It includes the following features:

- **Swagger editor**: It is an online editor for creating, editing, and testing OpenAPI specifications using an intuitive interface. Refer to **https://editor.swagger.io/**.
- **Swagger UI**: It is a user interface library that generates interactive documentation for APIs described through OpenAPI. This documentation is generated automatically from the OpenAPI specification and can be integrated directly into web applications. It has the same GUI as the Swagger Editor.
- **Swagger Codegen**: It is a tool that generates **Software Development Kit** (**SDK**) client code, mock servers, and other useful components based on the OpenAPI specification. This plugin exists for a variety of programming languages.

*Figure 1.10* shows a screenshot of Swagger Editor, after copying an OpenAPI file in YAML format into its editor:

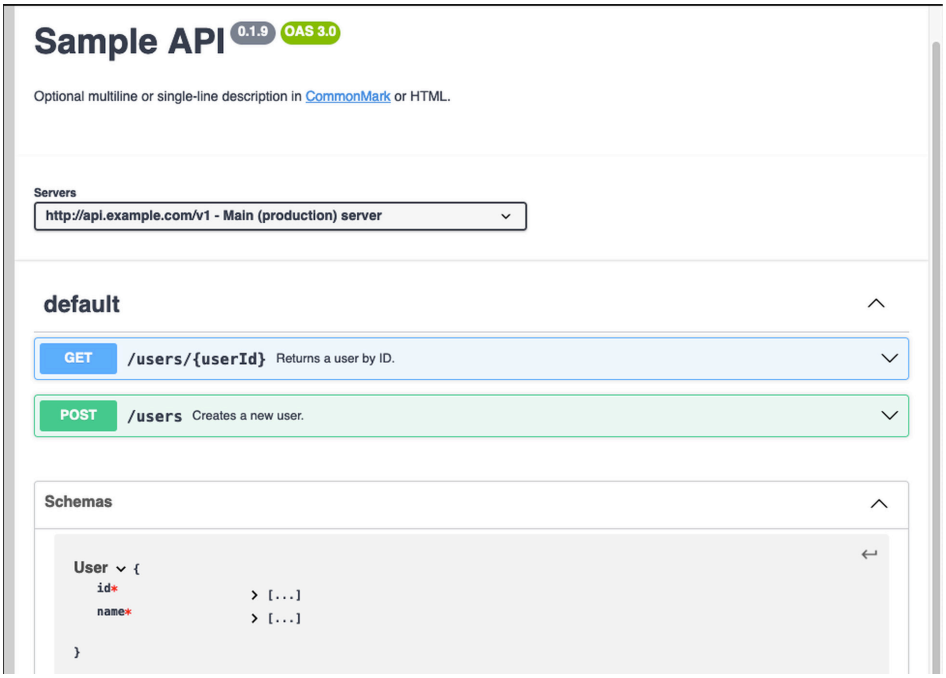



***Figure 1.10**: A screenshot of the Swagger Editor tool*

# API-first approach

In a traditional approach, we tend to start with the actual implementation of the software without detailed API design. The priority is on code development, with documentation and API design following afterwards. With this approach, changes to APIs may require significant changes to the existing code with the risk of causing compatibility problems.

The API-first approach aligns the API design phase with the design phase of the **Software Development Life Cycle (SDLC)**, treating the API definition as a core component from the outset. Rather than considering API design as a secondary activity or a later implementation, the API-first approach involves defining API specifications in detail before starting the actual software implementation. This ensures that the API design is robust and supports the application architecture effectively. The main advantages of this approach are mentioned below:

- By prioritizing API design, developers can focus on writing well-documented APIs that can easily evolve over time.
- It promotes collaboration among development teams, analysts, and project stakeholders.
- The API consumers will have documentation to work with immediately, without the need to wait for software developments. This is very important in a microservice architecture as it allows for parallelization of developments between services.
- At the stage after the OpenAPI documentation is written, developers can concentrate solely on the software development of the API, as it is already well defined.
- Developers can use the swagger plugin to automatically generate code, such as classes representing API resources.

# Conclusion

In this chapter, we have seen what an API is, and the different types of communication between microservices, with more emphasis on the REST architecture. Now that we have a solid foundation for writing APIs that fully respect the REST architecture, we must put it into practice by implementing REST services with Java and Spring Boot. In the next chapter, we will see how to implement this approach in a Spring Boot application through the OpenAPI Generator library.

# Points to remember

- Microservices communicate via APIs.
- The type of communication can be synchronous or asynchronous.

- REST API is the most widely used synchronous communication.
- Event-based is the most widely used asynchronous communication.
- REST API is based on the concepts of resource and HTTP verbs.

# Exercises

Try creating an OpenAPI file for REST services used by an e-commerce application. Then check its correctness in Swagger Editor.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

**https://discord.bpbonline.com**



# Chapter 2

# Reactive REST APIs with Spring WebFlux

## Introduction

In the modern landscape of microservices architecture, responsiveness, and scalability are paramount considerations for designing resilient **application programming interfaces** (**APIs**) that can handle diverse workloads and maintain optimal performance under varying conditions. In our second chapter, *Reactive REST APIs with Spring WebFlux*, we will explore reactive programming and its application in crafting high-performance **Representational State Transfer** (**REST**) APIs by using the Spring WebFlux. The chapter commences with a comprehensive introduction to the reactive paradigm, shedding light on its fundamental principles and the underlying philosophy of reactive programming. You will gain insights into the core concepts of the reactive framework, including Reactive Streams, publishers, and subscribers. A critical aspect of understanding reactive programming lies in distinguishing between the traditional event loop model and the thread-per-request model. You will explore these models in detail eventually. With the foundational knowledge in place, you are guided through the practical implementation of reactive REST APIs by using the Spring WebFlux and the API-first approach. Real-world examples demonstrate how to leverage Spring WebFlux to design and develop APIs that embrace reactive principles, enabling asynchronous and non-blocking interactions. You will also discover how to centralize error handling and request and response logs in Spring WebFlux. Furthermore, the chapter introduces WebClient, a powerful tool for building reactive REST clients, allowing developers to consume external services in a reactive, non-

blocking manner. Testing is an integral aspect of developing robust and reliable APIs, and our chapter concludes with a focus on testing reactive APIs.

# Structure

In this chapter we will discuss the following topics:

- Introduction to reactive paradigm
- Core concepts of reactive framework
- Event loop vs thread-per-request
- REST APIs with Spring WebFlux
- Reactive REST client with WebClient
- Testing reactive APIs
- Spring MVC vs Spring WebFlux performance

# Objectives

By the end of this chapter, you will have a total understanding of the reactive programming paradigm, and learn about Java's reactive API and the implementation of these with the Reactor library, which is the library used by Spring WebFlux to apply the reactive paradigm. You will also see the differences between the thread-per-request and event loop models used by Spring MVC and Spring WebFlux, respectively. With intelligent thread management in the event loop model, you can handle more HTTP requests with fewer computational resources. After a theoretical approach, you will learn how to write scalable and reactive REST APIs with Spring WebFlux and use the API-first approach with the OpenAPI Generator plugin. You will also see how error handling, logging, and testing work with this framework.

# Introduction to reactive paradigm

In recent years, the reactive paradigm (also known as the *reactive programming paradigm*), has emerged as an ideal approach for building highly reactive, scalable, and resilient software systems. Unlike traditional imperative programming models, which follow a sequential execution flow, reactive programming embraces a fundamentally different philosophy, focusing on asynchronous and event-driven architectures.

The reactive paradigm revolves around the concept of reacting to changes or events in the system, rather than explicitly controlling the flow of execution. This approach is particularly suitable for handling data streams or events with unpredictable arrival times or frequencies. This paradigm is especially ideal for cloud native applications, that is, applications that not only are deployed in cloud environments but also take advantage of all the power and features of cloud computing.

The key principles of the paradigm are well described in the *Reactive Manifesto* **https://www.reactivemanifesto.org**. A responsive system has the following characteristics:

- **Responsive**: A responsive system responds to the client in a fast and timely manner. It is usable even under critical conditions. If a problem occurs, the system must be able to immediately identify and isolate it, keeping its responsiveness intact.
- **Resilient**: A responsive system is resilient to failures. This aspect is related to the characteristic of responsiveness. A system that is not fault-tolerant will not be able to be a consistently responsive system. A responsive system implements resilience through component replication and fault isolation. The fault is isolated to the affected component only, while the other components respond in a way that does not create disruption. An external delegated component is responsible for recovering the component affected by the failure and ensuring high availability of the system through replication. This allows the clients to relieve themselves of the responsibility of having to deal with server system failures.
- **Elastic**: A reactive system is elastic to varying workloads. That is, it adapts quickly to changes in the frequency of incoming inputs, increasing or decreasing the resources allocated by horizontal scaling. Further, the horizontal scaling allows the workload to be divided among different instances. They also, through predictive algorithms, always maintain a suitable number of instances to effectively manage the workload but also be efficient in terms of resources used (and paid for).
- **Message-driven**: In a reactive system, components communicate through the exchange of events or messages, triggering reactions and transformations throughout the system. The event-driven architecture promotes reduced coupling between components and facilitates scalability and fault tolerance. These systems operate asynchronously, allowing components to perform non-blocking operations and respond to events as they occur. The asynchronous nature facilitates efficient resource utilization and allows systems to handle large numbers of simultaneous requests without compromising responsiveness. Responsive systems incorporate mechanisms to manage backpressure, allowing downstream components to control the rate of data or event consumption based on their processing capabilities. This ensures that resources are used optimally and prevents overload scenarios.

*Figure 2.1* summarizes the characteristics of a reactive system for the Reactive Manifesto:

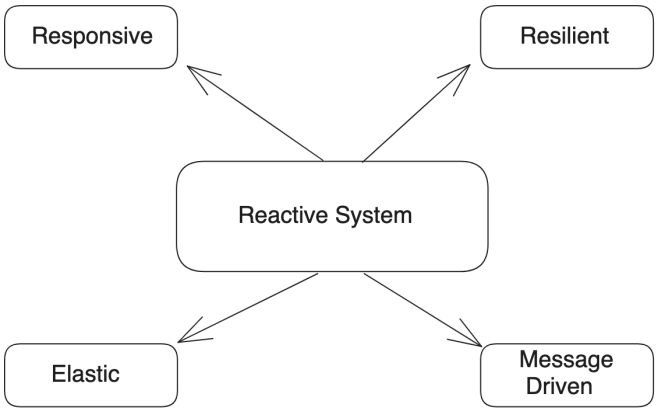



***Figure 2.1**: The features of a reactive system described in the Reactive Manifesto*

By embracing these principles, developers can design systems that are more resilient, scalable, and responsive to changing conditions and user demands. In the context of microservices architecture, the responsive paradigm offers significant advantages, enabling services to handle workloads in an optimized manner, saving both computational resources and economic.

Speaking of the asynchronicity of reactive programming, you may feel confused about the concepts of synchronous and asynchronous communication. Let us clarify these points well. Synchronous communication and asynchronous communication are techniques concerning the type of communication between services. Reactive programming is a paradigm concerning how to write and engineer a service. You can adopt the reactive paradigm either to implement REST APIs (synchronous communication) or to implement the publish/subscribe pattern (asynchronous communication). In this chapter, we will focus on creating REST APIs with the reactive paradigm by using the Spring WebFlux.

Many concepts from the Reactive Manifesto, such as horizontal scaling, are also echoed in microservice architecture and container technology. This is because all three have a common denominator, the optimal creation of a cloud native application. Although container technology existed before the development of microservices architecture, it has been used in enterprise projects precisely because of the microservices architecture. The rapid deployment of this technology enabled the development of projects such as Docker and Kubernetes that facilitated container management.

# Core concepts of the reactive framework

The Reactive Streams specification is a major initiative that aims to provide a standard for managing asynchronous, reactive data streams in Java. This specification was developed to address some of the challenges associated with managing data streams in a reactive,

non-blocking manner by providing a common interface and rule set that can be used by different reactive frameworks and libraries.

An important goal of the Reactive Streams specification is compatibility among different reactive frameworks and libraries. By adopting a common interface for handling data streams, the specification allows different reactive frameworks to interoperate smoothly with each other, enabling developers to combine different technologies and libraries within their applications. The Reactive Streams specification has been available since Java version 9 has been widely adopted in the Java development world and has helped standardize the approach to reactive programming. Reactive frameworks, including Reactor, RxJava, and Akka Streams, support the Reactive Streams specification and adopt its core principles to ensure effective and scalable management of asynchronous and reactive data streams. Spring WebFlux uses the Reactor library to implement the reactive paradigm on REST services. The reactive paradigm is considered an extension of the observer design pattern.

The interfaces provided by Reactive Streams are as follows:

```
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> var1);
}

public interface Subscriber<T> {
    void onSubscribe(Subscription var1);
    void onNext(T var1);
    void onError(Throwable var1);
    void onComplete();
}

public interface Subscription {
    void request(long var1);
    void cancel();
}

public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

Let us explain the responsibilities of the interfaces listed above:

- **Publisher**: It represents the data source within a responsive stream. It is responsible for issuing data elements to subscribers in response to subscription requests. The publisher can issue any number of data elements, or it can complete the flow when there are no more data elements to issue.

- **Subscriber**: It represents the data consumer within a responsive flow. It subscribes to a publisher to receive data elements issued by them, via the **`onSubscribe()`** method and a subscription. The subscriber receives data elements via the **`onNext()`** method, handles any errors via the **`onError()`** method, and signals the completion of the flow via the **`onComplete()`** method.
- **Subscription**: It represents the subscription contract between a publisher and a subscriber. It is responsible for managing communication between the two, allowing the subscriber to request data elements from the publisher and to manage backpressure by telling it its processing capability, via the **`request()`** method. Subscription also allows the subscriber to cancel the subscription at any time, interrupting the flow of data, via the **`cancel()`** method.
- **Processor**: It represents an entity that acts as both publisher and subscriber within a responsive flow. It receives data elements from a publisher, processes them according to its business logic, and sends them to the associated subscribers. The processor plays a key role in the transformation and processing of data streams within a reactive system.

Let us now focus on the Reactor library. Reactor implements the publisher interface with two classes, Mono and Flux. A Mono object represents a data stream that will output one element, while a Flux object represents a data stream that can output zero or more elements over time.

| Cardinality | Sync version | Async version |
|---|---|---|
| 0..1 | Optional<T> | Mono<T> |
| 0..N | Collection<T> | Flux<T> |

***Table 2.1:*** *A comparison of sychronous and asychronous objects in Java*

Operators are functions that allow data streams to be manipulated and transformed in different ways. Reactor provides a wide range of pre-defined operators that allow you to filter, map, reduce, and combine data responsively and efficiently. These operators are critical for creating complex data streams and managing reactive operations within applications. Among the most important are:

- **filter()**: It filters the elements of a stream based on a predicate.
- **map()**: It transforms each element of the flow by applying a synchronous function.
- **flatMap()**: It transforms each stream element asynchronously, accepting a function that in turn returns a new Mono/Flux.
- **reduce()**: It allows you to aggregate the elements of a Flux by using a specified reduction operation. This operator is useful when you want to combine all the elements of a stream into a single result (e.g., in a stream of integers, you want to sum them).

Here is an example of using these operators:

```
void testSum() {
    Flux.just(1, 2, 3, 4)
            .filter(integer -> integer % 2 == 0)
            .map(eventInteger -> eventInteger * 2)
            .flatMap(evenIntegerDoubled ->
                    Mono.fromSupplier(() -> evenIntegerDoubled + 1)
            )
            .reduce((firstInteger, secondInteger) ->
                    firstInteger + secondInteger
            )
            .doOnSuccess(finalInteger ->
                    log.info("sum: {}", finalInteger)
            )
            .subscribe();
}
```

Let us analyze the code:

- A stream of integers is created with the **`Flux.just()`** method.
- Only even integers are filtered with the **`Flux.filter()`** method.
- Each filtered integer is doubled by a synchronous function, with the **`Flux.map()`** method.
- For each doubled integer, an asynchronous function is applied that increments the integer by one unit, with the **`Flux.flatMap()`** method.
- The sum of all the doubled integers with the **`Flux.reduce()`** method is performed.
- After the last integer is also summed, it is printed by using the **`Flux.doOnSuccess()`** method. In this example, 14 is printed.
- Finally, the stream is subscribed with the **`subscribe()`** method. This is because the publisher (in this case the Flux), does not perform operations until it is subscribed.

From the code listed above, you will notice a different programming style adopted from the traditional one where, the imperative paradigm is used, and the program is organized with a set of instructions that describes step by step how to perform a given operation.

The functional paradigm, which is a type of declarative paradigm, is based on the execution of functions. They receive a value as input and return a new value as output. This goes well with the concept of immutability, precisely because the input object is never changed.

done, rather than how it should be done. In this paradigm, the programmer declares what operations or transformations should be performed, without specifying details about how these operations should be implemented. The concept of a functional paradigm has already been introduced in the Stream API of Java 8.

Another important concept in the Reactor library is the scheduler. Most operators continue to work on the same thread in which the previous operator was executed. Unless otherwise specified, the topmost operator, i.e., the data source, is executed in the same thread in which the `subscribe()` method was called. However, the library allows developers to choose different concurrency management strategies by using schedulers.

A scheduler can be thought of as an object similar to `java.util.concurrent.ExecutorService`, but dedicated to the reactive paradigm. The schedulers class has static methods for creating different schedulers:

- **Schedulers.immediate()**: This scheduler executes operations on the same thread as the caller. It is mostly used on APIs that require a scheduler as an input, but do not require execution on new threads.
- **Schedulers.single()**: This scheduler uses a single thread to execute all scheduled operations. It is particularly useful for operations that must be executed sequentially or that require access to shared resources, thus avoiding concurrency.
- **Schedulers.boundedElastic()**: This scheduler uses a thread pool with a pre-defined upper limit to execute scheduled operations. It is particularly useful for blocking operations, such as **Input/Output** (**I/O**) operations, so that other resources are not kept busy. As of Reactor version 3.6.0, it has two implementations, one that works on platform threads and one that works on virtual threads.
- **Schedulers.parallel()**: This scheduler uses a pool of dedicated threads to perform operations in parallel. The number of threads in the pool can be specified, allowing you to control the desired level of parallelism. It is useful for operations that can be executed independently and can benefit from parallelization.

Some operators use specific schedulers by default, such as `Flux.interval()`, `Flux.delayElement()`, `Mono.delay()` which use `Schedulers.parallel()`.

In Reactor, two operators allow changing the context execution via scheduler, publishOn, and subscribeOn. Let us see the differences:

- **publishOn()**: It is used to specify the execution context for subsequent operations in the data stream. This means that all operations after the `publishOn` operator will be executed on a thread associated with the scheduler specified by publishOn. This can be useful for changing the execution context for computationally expensive operations on a separate thread, thus avoiding blocking the main thread.
- **subscribeOn()**: It is used to specify the execution context for the entire data stream, including subscription operations and subsequent operations in the stream. This

means that all operations in the flow, including the initial subscribe operation, will be executed on a thread associated with the scheduler specified by subscribeOn. This can be useful for specifying the global execution context for the flow, for example, to execute all operations in the flow on a separate thread, thus ensuring consistent and predictable behavior of the flow.

The main difference between the two operators is when they affect the execution context of the data stream. The first, publishOn, affects only subsequent operations in the stream, while the second, subscribeOn, affects the entire data stream, including the initial subscribe and subsequent operations in the stream. Another difference is that publishOn can be used multiple times within a stream to change the execution context at specific points, while subscribeOn is generally used once at the beginning of the stream to specify the global execution context for the entire stream.

We modify the previous example by adding schedulers and logs to help us understand the execution context switch:

```
void testSum() {
    Flux.just(1, 2, 3, 4)
            .subscribeOn(Schedulers.boundedElastic())
            .filter(integer -> integer % 2 == 0)
            .doOnNext(eventInteger ->
                    log.info("Filtered integer: {}", eventInteger)
            )
            .map(eventInteger -> eventInteger * 2)
            .flatMap(evenIntegerDoubled ->
                    Mono.fromSupplier(() -> evenIntegerDoubled + 1)
            )
            .publishOn(Schedulers.parallel())
        .doOnNext(finalInteger -> log.info("Final integer: {}", finalInteger))
            .reduce((firstInteger, secondInteger) ->
                    firstInteger + secondInteger
            )
            .doOnSuccess(finalInteger ->
                    log.info("sum: {}", finalInteger)
            )
            .subscribe();
}
```

Let us look at the logs after the execution of this method:

```
20:05:05.818 [boundedElastic-1] INFO com.example.Test – Filtered integer: 2
20:05:05.819 [boundedElastic-1] INFO com.example.Test – Filtered integer: 4
20:05:05.820 [parallel-1] INFO com.example.Test – Final integer: 5
20:05:05.820 [parallel-1] INFO com.example.Test – Final integer: 9
20:05:05.820 [parallel-1] INFO com.example.Test – sum: 14
```

**Note: The printing of the result of the filter operator is done on a Schedulers.boundedElastic thread, this is because we used the subscribeOn(Schedulers.boundedElastic()) instruction. If we had not used that instruction, the execution of the chain (before publishOn) would have taken place on the main thread. Subsequent prints are done on a Schedulers.parallel thread, this is because we inserted the publishOn(Schedulers.parallel()) instruction after the flatMap method. So, all statements after the flatMap method will experience a change in the execution context.**

# Event Loop vs. Thread-per-request

Let us see how the reactive paradigm is used for handling HTTP requests. In applications using a synchronous Servlet Container, the thread-per-request model is applied, where for each incoming HTTP request, a thread from a thread pool managed by the Servlet Container is allocated. The thread is responsible for handling the entire lifecycle of the request, including processing, I/O operations, and response generation. *Figure 2.2* shows how the thread-per-request model works; each HTTP request occupies a different thread within a thread pool. The thread is released only when the request finishes its processing.

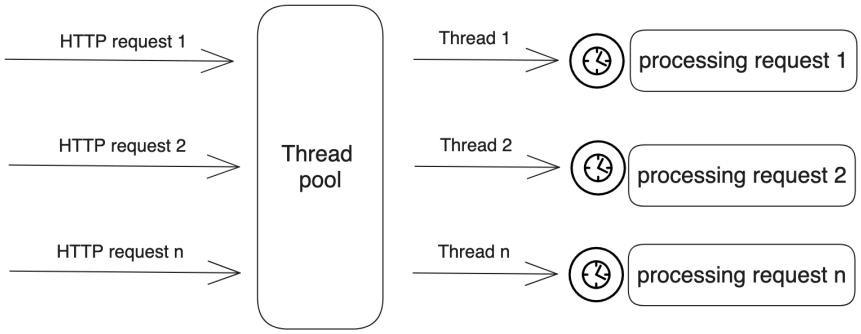



***Figure 2.2**: The thread-per-request model*

In intensive processing such as I/O operations, the thread may wait a long time until the operation finishes. The number of threads in a thread pool is limited. For example, in Apache Tomcat, the default number is 200. This means that if the application received 201 simultaneous HTTP requests, 200 would be processed in parallel, request number 201

requests in parallel, you could modify the configuration of the Servlet Container, increasing the number of threads in the pool. Increasing the number of threads, however, results in having to use more CPU, Java threads being mapped one-to-one with platform threads. More processing in the cloud means more overhead. We will see in the next chapter how virtual threads have changed the concept of threads within the Java ecosystem. The Spring MVC framework uses the thread-per-request model.

In the event loop model, used by non-blocking reactive frameworks such as Netty and Spring WebFlux, a single thread can handle multiple HTTP requests. Each request is placed in an event queue. The main thread, the one in the event loop, peels events from the queue as it goes and processes them. If it encounters a blocking operation in the processing, it registers a callback on a dedicated thread without blocking the main thread, so more events are queued from the queue. When the blocking operation ends, the callback returns the result to the event loop. In this mode, you have no blocked threads waiting for a result, which leads to several advantages:

- **Scalability**: The event loop can handle many simultaneous requests by using fewer threads.
- **Resource efficiency**: The event loop avoids wasting resources that are waiting for the blocking operations to complete.
- **Cost efficiency**: Being able to use fewer resources to handle more requests inevitably results in cost savings, since in the cloud you pay for what you use.

*Figure 2.3* shows the operation of the event loop model:

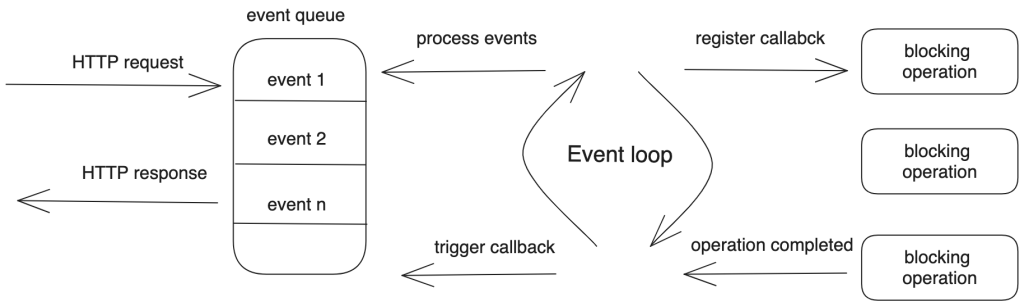



***Figure 2.3**: HTTP request handling in the event loop paradigm*

# REST APIs with Spring WebFlux

In the previous chapter, we introduced the notions of reactive paradigm and event loops, now you are ready to put these concepts into practice with Spring WebFlux. Throughout the chapters, we will create microservices for an e-commerce application called *Easyshop* that sells laptops and smartphones. We will use Java version 21, along with Spring Boot version 3, and build the projects by using Apache Maven. We will also use the Docker

**Integrated Development Environment** (**IDE**), although IntelliJ IDEA is recommended. The source code is available in the GitHub repository at the following link: **https://github.com/bpbpublications/Spring-Boot-3-API-Mastery-Advanced-Techniques-for-Microservices**. The repository is composed of folders representing each chapter where we will either be creating new microservices or modifying existing ones.

# Installation of Java 21

In order to install any version of Java, you can use several ways. The installation mode that is recommended is through the **Software Development Kit Manager** (**SDKMAN**) tool. This tool allows you to install different Java versions and easily change the current version to use with a simple command, without the need to manually set the JAVA_HOME environment variable. In addition, it is a cross-platform tool, so you can use it with Linux, Mac, or Windows.

In order to install the tool, open a **Command Line Interface** (**CLI**) and run the following command, `curl -s "https://get.sdkman.io" | bash`. After that, open a new CLI and check whether the installation was successful with the `sdk version` command. If there were no errors, you should see the version of SDKMAN installed. Now, you can install Java 21.

There are **Java Development Kits** (**JDKs**) from different providers, such as OpenJDK, Oracle, and AWS. You can choose any provider. With the `sdk list java` command you can see the latest versions of JDKs made available by all providers. To install one of these versions, run the `sdk install java <identifier>` command, for example, to install Oracle's JDK 21, the command is as follows, `sdk install java 21.0.2-oracle`. If the installation was successful, the `javac -version` command should display the version of the JDK installed. If you would like to learn more about SDKMAN, visit **https://sdkman.io/usage**.

# Installation of Docker Desktop

Docker Desktop is a cross-platform tool that allows both the client and server parts of Docker to be installed in a virtualized manner so that it can be used not only for Linux but also for Mac and Windows. In addition, the tool also provides a user interface to help manage containers. If you use Linux, consider using Docker natively instead of Docker Desktop. Docker is essential for creating and managing containers. Rather than installing the databases and queues locally, we prefer that you create the containers for these tools, so you can easily create and destroy them with a simple command. Also, in the final chapter, we will see how to containerize our microservices. You can easily install Docker Desktop from the official website, **https://www.docker.com/products/docker-desktop**.

# Installation of HTTPie

To invoke the HTTP endpoints we will create, I will use an open-source command-line tool called **HTTPie**, which compared to the **cURL** tool, is easier to use and formats out-of-box HTTP responses in a clearer format. The tool is cross-platform and is easily installed from the command line. To learn more, visit **https://httpie.io/docs/cli/installation**. Feel free to use any HTTP tool, however, this one is used in the book.

# Set up the Spring WebFlux project

Now that we have all the necessary tools at our disposal, we can write our first microservice, called catalog service. This microservice is responsible for managing the master data of the products that the e-commerce offers for sale.

In order to start a project with Spring Boot from scratch, you should use the Spring Initializr site available at this link, **https://start.spring.io**. From the web interface, you can choose the version of Spring Boot, the version of Java, and which Spring modules to use.

*Figure 2.4* shows what the Spring Initializr interface looks like and what Spring modules we import for this microservice:

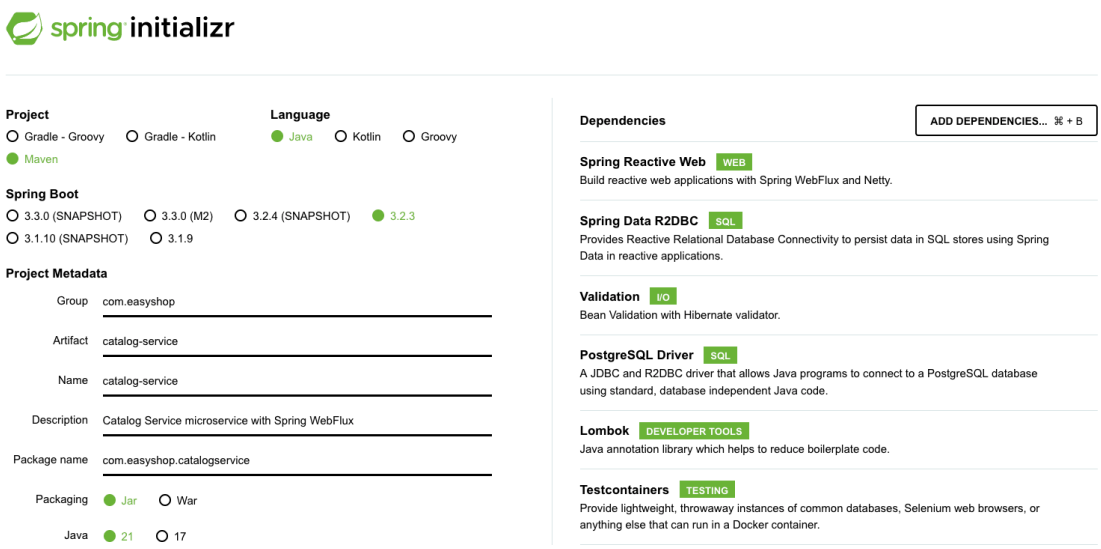



***Figure 2.4**: The graphical user interface of Spring Initializr*

Let us analyze the imported modules:

- **Spring Reactive Web:** It allows you to import the Spring WebFlux module which is needed to write REST services with the reactive paradigm.
- **Spring Data R2DBC:** It is a module of Spring Data that allows you to interface

WebFlux, you should handle everything reactively, including interactions with the database.

- **Validation:** It allows importing the spring-boot-starter-validation module, which is useful for validating fields on objects, via Java validation annotations.
- **PostgreSQL Driver**: It allows importing the Postgres driver to establish database connections. With the presence of the Spring Data R2DBC module, Spring Initializr will import the reactive Postgres driver, i.e., the R2DBC one, and not the traditional one, i.e., JDBC. We chose to use Postgres because it is the most widely used relational database. In addition, all cloud providers make available managed services with this type of database.
- **Lombok:** It is a useful library that allows you to autogenerate methods such as getter, setter, equals, and hashcode, useful for keeping code cleaner.
- **Testcontainers:** It is a library that allows us to write integration tests by using a containerized environment such as Docker. For example, to test a method of a repository class, we could use a Postgres database created for the test. When the test ends, the Postgres container is automatically deleted from the library. Useful for performing integration tests closer to the production environment.

In the file **`chapter-02/catalog-service/README.md`** we have included the Spring Initializr link that allows you to already have the configuration shown above.

Download the project, unzip it, and open it with an IDE. You will find the structure of the project done this way, as shown in *Figure 2.5*:

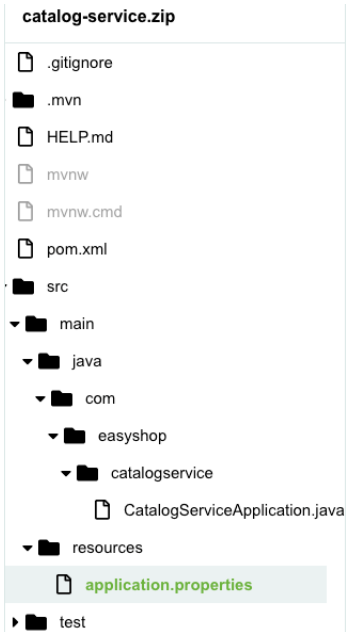



*Figure 2.5*

If you try to run the main class, you will see an error message, missing parameters to connect to the database.

# Designing the product domain object

Before a stable database connection, we design the domain object. For simplicity, we can say that a product consists of the following:

- A unique code that identifies the product.
- An optional name.
- A category, between laptop and smartphone.
- A price in a eurocent format.
- The name of a brand (optional).

**Note: Using an integer type (e.g., "long" in Java) for price representation, such as in cents, avoids rounding errors associated with floating-point types ('double") and ensures precise calculations, crucial for financial applications. This approach improves consistency, simplifies arithmetic operations, and aligns with best practices recommended by many payment systems, enhancing integration and reliability.**

Now that we have a good idea of how the domain object should be composed, we can create the database table. Start the Docker Desktop, then open a terminal and place yourself inside the **`chapter-02/infra/docker`** folder. In this folder, we would find the file **`docker.compose.yml`** that allows, via Docker Compose, to start Postgres locally. Let us take a look at the file:

```
version: '3'
services:
  pg:
    image: 'postgres:16.2'
    container_name: "easyshop-postgres"
    ports:
      - 5432:5432
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - ./postgres/init.sql:/docker-entrypoint-initdb.d/init.sql
      - easyshop-postgres:/var/lib/postgresql/data
volumes:
  easyshop-postgres:
```

**Note: The container uses two volumes. The first is used to initialize the database, via the init.sql script; the second is used to maintain data persistence even if we destroy and recreate the container.**

You can find the **`init.sql`** file inside the **`chapter-02/infra/docker/postgres`** folder, and it has the following content:

```
CREATE DATABASE easyshopdb_catalog;
\c easyshopdb_catalog;
CREATE TABLE IF NOT EXISTS products (
    product_id SERIAL PRIMARY KEY,
    code VARCHAR UNIQUE NOT NULL,
    name VARCHAR,
    price BIGINT,
    brand VARCHAR,
    category VARCHAR
);
```

The SQL script creates the **`easyshopdb_catalog`** database and on this, it creates the products table. Note that in addition to the fields we listed above, there is also the **`product_id`** field, which is the primary key, and is an auto-incremental integer. We could have chosen the **`code`** field as the key, which is unique, but we preferred an integer for simplicity. However, we need to hide this handling from the client. From the terminal, run the **`docker compose up -d`** command to start the containers. Now, we have started our database. In the **`docker.compose.yml`** file on GitHub, you will also find the **`pgadmin`** container declaration, to display through a user interface the contents of the database. Feel free to use this or another tool of your choice.

# Getting started with Spring WebFlux

The APIs we will implement will be used to handle CRUD operations on products. Let us begin by designing the API for inserting a new product. *Figure 2.6* shows the sequence diagram of the classes involved:

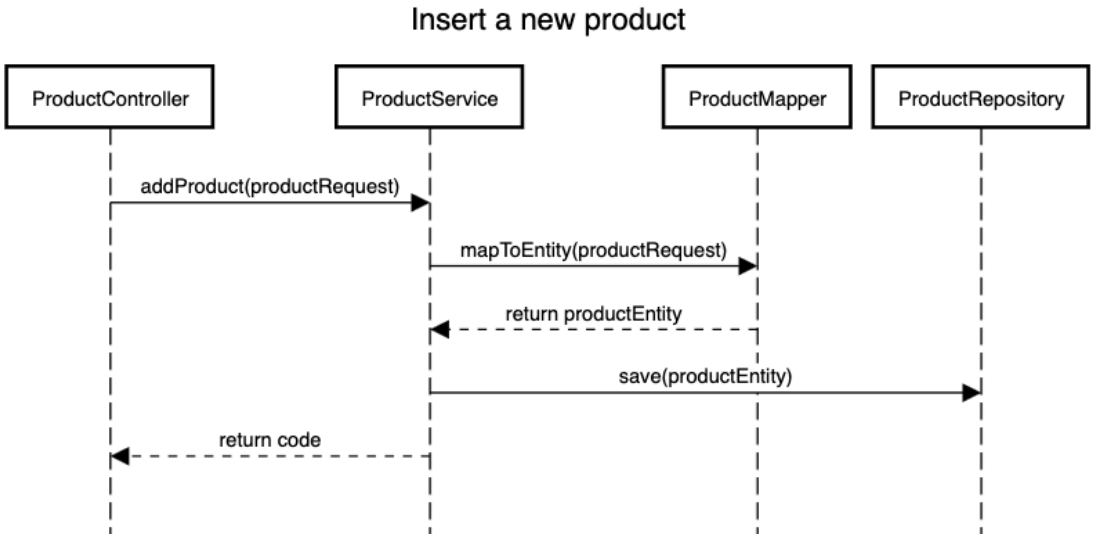



***Figure 2.6**: The sequence diagram of the operation of inserting a new product*

As we use the API-first approach, we write the OpenAPI file first. Create the **`openapi/server`** folder inside the project root folder and create the **`catalog-service.yml`** file; this file will contain the OpenAPI specification. Let us design the product creation endpoint:

```
paths:
  /products:
    post:
      tags:
        - product
      summary: Add a new product to the store
      operationId: addProduct
      requestBody:
        description: Create a new product in the store
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ProductRequest'
        required: true
      responses:
        '201':
          description: Successful operation
```

```
        headers:
          Location:
            required: true
            schema: string
            format: uri
      '400':
        description: Invalid input
        content:
          application/problem+json:
            schema:
              $ref: '#/components/schemas/ProblemDetail'
```

The creation of a new product is mapped with a POST API that returns HTTP status code 201 with the **`Location`** header containing the **Uniform Resource Identifier** (**URI**) of the created resource. Note that for an invalid request, HTTP status code 400 and a response body of type **`ProblemDetail`** is returned. Let us look at the request and response bodies in detail:

```
components:
  schemas:
    ProductRequest:
      required:
        - code
        - category
        - price
      type: object
      properties:
        code:
          type: string
          example: Product Code
        name:
          type: string
          example: Product Name
        category:
          $ref: '#/components/schemas/ProductCategory'
        price:
          type: integer
```

```
          format: int64
          example: 1000
        brand:
          type: string
          example: TopComputer
    ProductCategory:
      type: string
      enum:
        - laptop
        - smartphone
    ProblemDetail:
      type: object
      properties:
        type:
          type: string
          format: uri
        title:
          type: string
        status:
          type: integer
          format: int32
        detail:
          type: string
```

The request, of type **`ProductRequest`**, contains all the fields of the domain object. As for the **`ProblemDetail`** schema, it is a container that details a problem and is represented by RFC 7807 (**https://datatracker.ietf.org/doc/html/rfc7807**). It is good practice to associate this object with any HTTP status related to a problem, with content type **`application/problem+json`**. With the Spring Boot 3 release, we already have the **`org.springframework.http.ProblemDetail`** class that maps the **`ProblemDetail`** to RFC 7807.

Now that we have written the OpenAPI file, we autogenerate the Java classes and use the OpenAPI Generator plugin. We chose to use this plugin out of many because it is an open-source project with a very active community. Open the **`chapter-02/catalog-service/pom.xml`** file and paste the following code in the plugins section to use the OpenAPI Generator plugin:

```
<plugin>
    <groupId>org.openapitools</groupId>
```

```
    <artifactId>openapi-generator-maven-plugin</artifactId>
    <version>7.4.0</version>
    <executions>
      <execution>
         <goals>
             <goal>generate</goal>
         </goals>
         <configuration>
          <inputSpec>
               ${project.basedir}/openapi/server/catalog-service.yml
          </inputSpec>
         <openapiNormalizer>REF_AS_PARENT_IN_ALLOF=true</openapiNormalizer>
          <generatorName>spring</generatorName>
         <apiPackage>com.easyshop.catalogservice.generated.api</apiPackage>
               <modelPackage>com.easyshop.catalogservice.generated.model</
modelPackage>
          <schemaMappings>
              ProblemDetail=org.springframework.http.ProblemDetail
           </schemaMappings>
           <configOptions>
                  <interfaceOnly>true</interfaceOnly>
                  <delegatePattern>true</delegatePattern>
                  <useSpringBoot3>true</useSpringBoot3>
                  <openApiNullable>false</openApiNullable>
                  <useTags>true</useTags>
                  <reactive>true</reactive>
           </configOptions>
         </configuration>
        </execution>
    </executions>
</plugin>
```

Let us analyze the most important properties set in the plugin:

- The **`interfaceOnly`** property enabled causes the plugin to generate only API interfaces. It is the developer's responsibility to implement them with a class annotated with **`@RestController`**.

- With the **`useTags`** property enabled, the generator will create the name of the interfaces from the tags in the OpenAPI file. The method names of the interfaces will be equal to the **`operationId`** of the OpenAPI file.
- Since with version 3, Spring Boot already provides the **`ProblemDetail`** class, we avoid having the plugin generate it as well by using the **`schemaMappings`** property.

The OpenAPI Generator plugin uses Swagger annotations, so you need to import them into the project, for example by adding the following dependency:

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webflux-api</artifactId>
    <version>2.3.0</version>
</dependency>
```

With the terminal, place yourself in the root folder of the project and run the following command to autogenerate the classes: **`./mvnw clean compile`**. Let's take a look at the autogenerated classes, in the target folder. We have the request model:

```
public class ProductRequest {
    private String code;
    private String name;
    private ProductCategory category;
    private Long price;
    private String brand;

   //getters,setters...
}
```

Let's look at the interface that contains the API to be implemented:

```
@Validated
public interface ProductApi {

    @RequestMapping(
            method = {RequestMethod.POST},
            value = {"/products"},
            produces = {"application/problem+json"},
            consumes = {"application/json"}
    )
```

```
    default Mono<ResponseEntity<Void>> _addProduct(@RequestBody @Valid
                                                   Mono<ProductRequest>
                                                     productRequest,
                                                   final ServerWebExchange
                                                     exchange) {
        return this.addProduct(productRequest, exchange);
    }

    default Mono<ResponseEntity<Void>> addProduct(Mono<ProductRequest>
                          productRequest, final ServerWebExchange exchange) {
        //a default response
    }
}
```

**Note: The interface already contains the validation annotations, `@Validated` and `@Valid`, and the web annotations `@RequestMapping` and `@RequestBody`. The concrete class will need to implement the `addProduct` interface. The response object of the `addProduct` method is of type `ResponseEntity`, which is a Spring class that contains all the data of an HTTP response, such as the body and HTTP status. The object is wrapped in an object of type Mono because we are using Spring WebFlux.**

Before implementing the interface, let us write a class that maps the database table and products. Create the **`middleware.db.entity`** package and write the **`ProductEntity`** class:

```
@Table(name = "products")
public record ProductEntity(
        @Id Long productId,
        String code,
        String name,
        Long price,
        String brand,
        CategoryEntity category
) {}
```

We wrote this class by taking advantage of Java 17's records feature, which allows us to create immutable objects with constructor and automatic getters, settets, equals, and hashcode methods. We also annotated the class with **`org.springframework.data.relational.core.mapping.Table`** to indicate to Spring Data that this class maps to fields in the table named products. The **`org.springframework.data.annotation.Id`** annotation on a field indicates to Spring Data that the field is the primary key for the table.

Let us now write the repository class that performs CRUD operations on the products. Create the repo subpackage inside the **middleware.db** package and write the following interface:

```
public interface ProductRepository extends R2dbcRepository<ProductEnti-
ty, Long> {

}
```

It is an empty interface that you do not need to implement. Spring Data will automatically create a bean of type SimpleR2dbcRepository at runtime, that will implement that interface by using the Proxy pattern. This repository offers out-of-the-box common methods such as **findAll**, **findById**, **save**, and **deleteById**. In addition, you can write additional methods, always empty, by using the name convention **findBy<name_field>**. For example, if you added the **findAllByName** method, Spring Data would automatically implement the method that will query the name field. In addition, Spring Data offers many modules besides R2DBC, such as **Java DataBase Connectivity** (**JDBC**) and **Java Persistence API** (**JPA**); however, the interfaces always contain the same methods. You can therefore swap Spring Data modules without changing the business logic of the application.

The last step in connecting the application to the database is to give it the connection data. Go to the **application.properties** file and write the following lines:

```
spring.r2dbc.url=r2dbc:postgresql://localhost:5432/easyshopdb_catalog
spring.r2dbc.username=user
spring.r2dbc.password=password
```

We can write the **ProductMapper** class, which will be responsible for transforming the request object, of type **ProductRequest**, into an object that maps to the database table, of type **ProductEntity**. Again, we will not write this class manually, but use the **MapStruct** library, which allows us to autogenerate mapper classes. Import the library into the pom.xml by adding the dependency:

```
<dependency>
    <groupId>org.mapstruct</groupId>
    <artifactId>mapstruct</artifactId>
    <version>1.5.5.Final</version>
</dependency>
```

Then, add the library plugin in this way:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
```

```
            <annotationProcessorPaths>
                <path>
                    <groupId>org.mapstruct</groupId>
                    <artifactId>mapstruct-processor</artifactId>
                    <version>1.5.5.Final</version>
                </path>
                <path>
                    <groupId>org.projectlombok</groupId>
                    <artifactId>lombok</artifactId>
                    <version>${lombok.version}</version>
                </path>
                <dependency>
                    <groupId>org.projectlombok</groupId>
                    <artifactId>lombok-mapstruct-binding</artifactId>
                    <version>0.2.0</version>
                </dependency>
            </annotationProcessorPaths>
            <compilerArgs>
                <compilerArg>
                    -Amapstruct.defaultComponentModel=spring
                </compilerArg>
            </compilerArgs>
        </configuration>
    </plugin>
```

The code allows you to manage the **`MapStruct`** plugin in the presence of the Lombok library.

Now, all that remains is to write the **`ProductMapper`** interface. Create the package mapper and write the Java interface as follows:

```
@Mapper(componentModel = "spring")
public interface ProductMapper {
    ProductEntity toEntity(ProductRequest productRequest);
}
```

Run the **`./mvnw clean compile`** command to autogenerate the class that implements this interface. Let us look at the autogenerated class:

```
@Component
public class ProductMapperImpl implements ProductMapper {

    @Override
    public ProductEntity toEntity(ProductRequest productRequest) {
        if (productRequest == null) {
            return null;
        }

        String code = null;
        String name = null;
        Long price = null;
        String brand = null;
        CategoryEntity category = null;

        code = productRequest.getCode();
        name = productRequest.getName();
        price = productRequest.getPrice();
        brand = productRequest.getBrand();
        category = productCategoryToCategoryEntity(productRequest.getCategory());

        Long productId = null;

        ProductEntity productEntity = new ProductEntity(productId, code, name,
                price, brand, category);

        return productEntity;
    }
}
```

The library automatically figures out how to map fields between the two classes from the field names.

The time has come to implement the business logic of the product insertion operation. Create the **`service`** package and implement the **`ProductService`** class:

```
@Service
@Transactional(readOnly = true)
@RequiredArgsConstructor
@Slf4j
public class ProductService {

    private final ProductRepository productRepository;
    private final ProductMapper productMapper;

    @Transactional
    public Mono<String> addProduct(ProductRequest productRequest) {
        return insertNewProduct(productRequest)
                .map(ProductEntity::code);
    }

     private Mono<ProductEntity> insertNewProduct(ProductRequest productRe-
quest) {
        var productEntity = productMapper.toEntity(productRequest);
        return productRepository.save(productEntity)
                .doOnNext(entitySaved ->
                        log.debug("Product saved: {}", entitySaved));
    }
}
```

The **`addProduct`** method simply maps the request into an object of type **`ProductEntity`** to then be saved to the database by **`ProductRepository`**. Finally, it returns the product code to the caller.

**Note: We annotated the class with `@Transactional(readOnly = true)` so that by default, all methods in the class use the read-only transaction type. We then annotated the `addProduct` method with the `@Transactional` annotation since we need to use a write transaction here. The `@RequiredArgsConstructor` annotation allows us to autogenerate the constructor with the final fields. Finally, the @Slf4j annotation allows us to use a logger.**

The last step is to create the **`ProductController`** class that implements the **`ProductApi`** autogenerated interface. Create the **`api`** package and write the following class:

```
@RestController
@RequiredArgsConstructor
@Slf4j
public class ProductController implements ProductApi {

    private final ProductService productService;

    @Override
    public Mono<ResponseEntity<Void>> addProduct(Mono<ProductRequest>
                                                        productRequest,
                                                ServerWebExchange exchange) {
        return productRequest
                .flatMap(productService::addProduct)
                .map(productCode -> ResponseEntity
                  .created(URI.create(exchange.getRequest().getURI() + "/" +
                                productCode)).build());
    }
}
```

The **`addProduct`** method is very simple; all it does is call the service method of the same name and finally return HTTP status code 201 with the Location field value along with the URI of the newly created resource. Remember that the methods of a **`Controller`** class should never contain business logic.

Start the application main class and test the endpoint. From the terminal, make an HTTP request like this:

```
http :8080/products code=ABCD-0000 name="Super Laptop" category=laptop
price=1000 brand=FirstBrand
```

The command simply makes a POST request on localhost:8080 with the following body:

```
{
   "code":"ABCD-0000",
   "name":"Super Laptop",
   "category":"laptop",
   "price":1000,
   "brand":"FirstBrand"
}
```

The default port used in Spring WebFlux and Spring MVC is 8080. You should see the following response:

```
HTTP/1.1 201 Created
Location: http://localhost:8080/products/ABCD-0000
content-length: 0
```

Well, the resource has been successfully created. You can verify this by viewing the contents of the table with pgadmin, as shown in *Figure 2.7*:

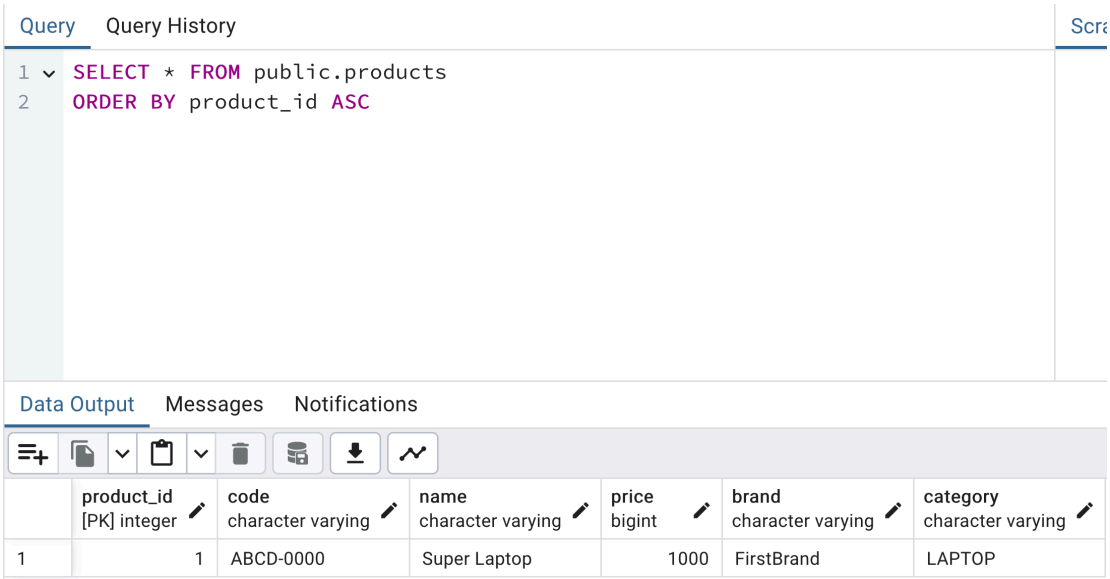



***Figure 2.7**: The screenshot of pgadmin showing the product you just entered*

Try to make a request that does not contain a required field, such as the following code:

```
http :8080/products name="Mega Laptop" category=laptop price=1000
brand=FirstBrand
```

The response will be the following:

```
HTTP/1.1 400 Bad Request
Content-Length: 124
Content-Type: application/json

{
```

```
    "error": "Bad Request",
    "path": "/products",
    "requestId": "27471c66-1",
    "status": 400,
    "timestamp": "2024-03-17T13:32:01.997+00:00"
}
```

Validation is performed automatically, returning 400 Bad Request. However, the response does not meet the **`ProbelmDetail`** specification.

Let us try making an insertion request with the same code. The product code is unique, so the API must respond with an error, 400 in this case. Make the same request that was successful:

```
http :8080/products code=ABCD-0000 name="Super Laptop" category=laptop
price=1000 brand=FirstBrand
```

The response will be as follows:

```
HTTP/1.1 500 Internal Server Error
Content-Length: 134
Content-Type: application/json

{
    "error": "Internal Server Error",
    "path": "/products",
    "requestId": "36eb129b-1",
    "status": 500,
    "timestamp": "2024-03-17T16:35:20.838+00:00"
}
```

The body and HTTP status are incorrect. Let us get these answers right for now. In the next section, we will show you how to handle errors with WebFlux in Spring Boot 3.

Let us now implement the endpoint of retrieving a product via its code, to verify that the URI of the **`Location`** header is correct. *Figure 2.8* shows the sequence diagram of the operation to retrieve a product via its code:

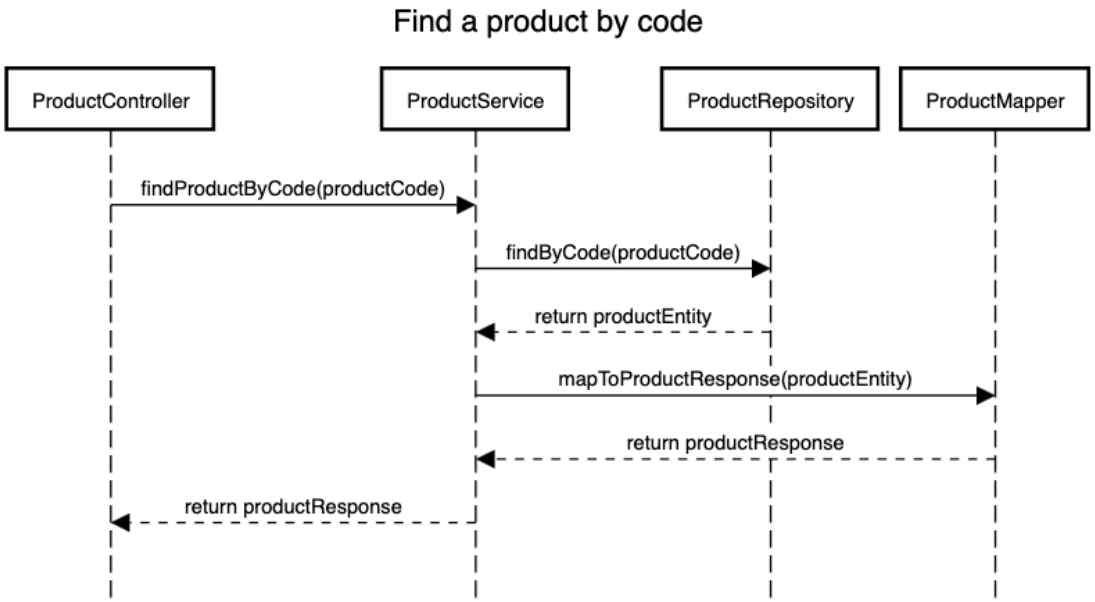



***Figure 2.8:*** *The sequence diagram of the operation of retrieving a product by its code*

Let us design the endpoint. Add this piece of code to the OpenAPI file:

```
/products/{productCode}:
  get:
    tags:
      - product
    summary: Find product by code
    description: Returns a single product
    operationId: findProductByCode
    parameters:
      - name: productCode
        in: path
        description: Code of productId to return
        required: true
        schema:
          type: string
    responses:
      '200':
        description: Get product by code
```

```
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ProductResponse'
      '404':
        description: Product not found
        content:
          application/problem+json:
            schema:
              $ref: '#/components/schemas/ProblemDetail'
```

The endpoint takes the input, via a URL parameter, the product code, and responds to 200 if the product was found, and 404 otherwise.

The **`ProductResponse`** schema could have the same fields as **`ProductRequest`**. Instead of copying the same fields, we can use the concept of model composition provided by the OpenAPI specification, via the keyword **`allOf`**. We can then create a parent schema, called **`BaseProduct`**, a child schema called **`ProductResponse`**, and modify the **`ProductRequest`** schema, in this manner:

```
ProductRequest:
  allOf:
    - $ref: '#/components/schemas/BaseProduct'
    - type: object
ProductResponse:
  allOf:
    - $ref: '#/components/schemas/BaseProduct'
    - type: object
BaseProduct:
  required:
    - code
    - category
    - price
  type: object
  properties:
    code:
      type: string
      example: Product Code
```

```
    name:
      type: string
      example: Product Name
    category:
      $ref: '#/components/schemas/ProductCategory'
    price:
      type: integer
      format: int64
      example: 1000
    brand:
      type: string
      example: TopComputer
```

Through this feature, **`ProductRequest`** and **`ProductResponse`** inherit **`BaseProduct`** fields. Now, we must decide whether to use inheritance on the Java side as well. By default, the OpenAPI Generator plugin, would not exploit the concept of inheritance but would copy the **`BaseProduct`** fields to the **`ProductRequest`** and **`ProductResponse`** classes. To take advantage of the concept of inheritance on Java, we add the configuration section of the plugin, in the **`pom.xml`** property:

```
<openapiNormalizer>REF_AS_PARENT_IN_ALLOF=true</openapiNormalizer>
```

We auto-generate the code with the usual Maven command and analyze the new model classes:

```
public class BaseProduct {

    private String code;
    private String name;
    private ProductCategory category;
    private Long price;
    private String brand;

    //getters, setters, etc
}

public class ProductRequest extends BaseProduct {
    //...
}
```

```
public class ProductResponse extends BaseProduct {
    //...
}
```

On Java, we have exploited inheritance. We implement the logic of the product retrieval operation. First, we add the following method to the repository class:

```
Mono<ProductEntity> findByCode(String code);
```

As we mentioned previously, Spring Data will automatically implement this method. Due to the name convention, the library knows that it must query by product code. We implement the business logic in **`ProductService`**:

```
public Mono<ProductResponse> findProductByCode(String productCode) {
    return productRepository.findByCode(productCode)
            .map(productMapper::toProductResponse);
}
```

The code contains just two lines; the query is made through the repository, which returns a **`ProductEntity`** and finally, the latter is transformed into a **`ProductResponse`**. The new method in the **`ProductMapper`** interface is as follows:

```
ProductResponse toProductResponse(ProductEntity productEntity);
```

Whenever you modify the OpenAPI file or the mapper, remember to autogenerate the Java classes again with the Maven command, **`./mvnw clean compile`**.

The last step is to implement the method in the **`ProductController`** class:

```
@Override
public Mono<ResponseEntity<ProductResponse>> findProductByCode(String pro-
ductCode, ServerWebExchange exchange) {
    return productService.findProductByCode(productCode)
            .map(ResponseEntity::ok);
}
```

This method also contains only two lines. It calls the business logic of the service and maps the result to the HTTP 200 status and response body. We test the endpoint, executing a GET request on the URI previously returned by the POST request:

```
http :8080/products/ABCD-0000
```

The answer you should have is:

```
HTTP/1.1 200 OK
Content-Length: 96
Content-Type: application/json

{
    "brand": "FirstBrand",
    "category": "laptop",
    "code": "ABCD-0000",
    "name": "Super Laptop",
    "price": 1000
}
```

The API returned the resource with the code ABCD-0000. Let us see what happens by requesting a non-existing code, **`http :8080/products/ABCDEF`**. The response will be as follows:

```
HTTP/1.1 200 OK
content-length: 0
```

The fact that the **`ProblemDetail`** is not mapped, is the least serious problem, the response even returned a 200 instead of a 400, making the client believe that there are no errors. Now, it is time to talk about exception handling in Spring WebFlux. In the GitHub repository, you will find the implementation of PUT and DELETE endpoints as well, in addition, you will also find the implementation of paged **`findAll`**.

# Exception handling in Spring WebFlux

The two endpoints implemented so far will work well if they follow the so-called *happy path* that is, if they follow the flow that does not lead to errors. In the POST, we have seen that in the case of a validation error on the request, 400 Bad Request is returned but the body does not meet the **`ProbelmDetail`** specification. In the case of entering a product with a code already censored, the API returns 500 instead of 400.

In the GET, if the input product code does not exist, the API returns a 200.

With Spring Boot 3, the WebFlux module provides a class that acts as an exception handler, **`ResponseEntityExceptionHandler`**. This class handles HTTP exceptions by returning a body that conforms to the **`ProblemDetail`** specification. The class can be automatically enabled by adding the following property in the **`application.properties`** file: **`spring.webflux.problemdetails.enabled=true`**.

However, to handle errors optimally, my advice is not to enable that property but to extend the Spring class, like this:

```
@RestControllerAdvice
@Slf4j
public class ErrorHandler extends ResponseEntityExceptionHandler {
}
```

By doing so, you get the same result as activating that property, but in addition, you can extend the class to handle custom exceptions.

For the GET API, we write a custom exception for the case of a product not found. Create the package *exception* and write the following class:

```
@Getter
public class ProductNotFoundException extends NestedRuntimeException {

    public ProductNotFoundException(String productCode) {
        super(String.format("Product with code %s not found", productCode));
    }
}
```

Now, use this class in the **`findProductByCode`** method of the **`ProductService`** class:

```
public Mono<ProductResponse> findProductByCode(String productCode) {
    return productRepository.findByCode(productCode)
       .switchIfEmpty(Mono.error(new ProductNotFoundException(productCode)))
            .map(productMapper::toProductResponse);
}
```

With the **`switchIfEmpty`** method of the **`Mono`** class, we handle the case when the repository does not find the product with the input code. In this case, we send an error signal of type **`ProductNotFoundException`**. We handle the exception in the **`ErrorHandler`** class by adding the following method:

```
@ExceptionHandler(ProductNotFoundException.class)
public Mono<ResponseEntity<Object>> handleConstraintViolation(
        ProductNotFoundException ex, ServerWebExchange serverWebExchange) {

    log.warn(ex.getMessage());
    var pd = ProblemDetail
```

```
    return Mono.just(new ResponseEntity<>(pd, HttpStatus.NOT_FOUND));

}
```

With the **`@ExceptionHandler`** annotation, we tell Spring which exceptions the method should handle, in this case, **`ProductNotFoundException`**. The method simply returns the correct HTTP status and the **`ProblemDetail`**. Let us re-execute the following HTTP request: **`http :8080/products/ABCDEF`**, the HTTP response will be:

```
HTTP/1.1 404 Not Found
Content-Length: 131
Content-Type: application/problem+json

{
    "detail": "Product with code ABCDEF not found",
    "instance": "/products/ABCDEF",
    "status": 404,
    "title": "Not Found",
    "type": "about:blank"
}
```

The API now handles this case correctly as well. For the POST API, we need to handle the cases where we enter the code that already exists and the case where the validation fails. For the second one, already enabling the exception handler will already fix the response with a **`ProblemDetail`**. To handle the first case instead, we create a custom exception:

```
@Getter
public class ProductAlreadyExistsException extends NestedRuntimeException {

    public ProductAlreadyExistsException(String productCode) {
     super(String.format("Product with code %s already exists", productCode));
    }
}
```

We use this exception in the **`addProduct`** method of the **`ProductService`** class:

```
@Transactional
public Mono<String> addProduct(ProductRequest productRequest) {
    return insertNewProduct(productRequest)
            .map(ProductEntity::code);
```

```
}

private Mono<ProductEntity> insertNewProduct(ProductRequest productRe-
quest) {
     final ProductEntity productEntity = productMapper.toEntity(productRe-
quest);
    return productRepository.save(productEntity)
            .doOnNext(entitySaved ->
                    log.debug("Product saved: {}", entitySaved))
            .onErrorResume(DuplicateKeyException.class, e ->
                    Mono.error(
                            new ProductAlreadyExistsException(productEntity.
code())
                    ));
}
```

The exception thrown by Spring Data in case of a duplicate unique field is **`DuplicateKeyException`**, so we handle the latter with the **`onErrorResume`** method of the Mono class and send an error signal with our custom exception. We handle the exception in our error handler class by adding the following method:

```
@ExceptionHandler(ProductAlreadyExistsException.class)
public Mono<ResponseEntity<Object>> handleConstraintViolation(
          ProductAlreadyExistsException ex, ServerWebExchange serverWebEx-
change) {


    log.warn(ex.getMessage());
    var pd = ProblemDetail
            .forStatusAndDetail(HttpStatus.BAD_REQUEST, ex.getMessage());
    return Mono.just(new ResponseEntity<>(pd, HttpStatus.BAD_REQUEST));
}
```

The method returns the status code 400 and the response body of type **`ProblemDetail`**. We executed the POST request that had given an error:

```
http :8080/products code=ABCD-0000 name="Super Laptop" category=laptop
price=1000 brand=FirstBrand
```

The answer this time is as follows:

```
HTTP/1.1 400 Bad Request
```

```
Content-Length: 134
Content-Type: application/problem+json

{
    "detail": "Product with code ABCD-0000 already exists",
    "instance": "/products",
    "status": 400,
    "title": "Bad Request",
    "type": "about:blank"
}
```

Now, the POST API handles this case correctly as well.

# Logging request and response in Spring WebFlux

It can be useful to log all HTTP requests and responses from our application for debugging purposes. You have several ways to do this, such as by implementing the WebFilter interface. However, the method we suggest, if you have no special requirements, is the simplest one, enable the HTTP detail logs made available by Spring WebFlux. In the **`application.properties`** file, add the following properties:

```
logging.level.org.springframework.web.server.adapter.HttpWebHandlerAdapter=TRACE
logging.level.org.springframework.web.HttpLogging=TRACE
```

Restart the application and try making a POST request, for example:

```
http :8080/products code=HILM name="HZ Laptop" category=laptop price=1000 brand=FirstBrand
```

Take a look at the IDE console, you should find the following logs:

```
o.s.w.s.adapter.HttpWebHandlerAdapter    : [49f91090-1] HTTP POST "/products", headers={masked}
org.springframework.web.HttpLogging      : [49f91090-1] Decoded [class ProductRequest {
    class BaseProduct {
        code: HILM
        name: HZ Laptop
        category: laptop
        price: 1000
        brand: FirstBrand
```

```
    }
}]
o.s.w.s.adapter.HttpWebHandlerAdapter    : [49f91090-1] Completed 201 CRE-
ATED, headers={masked}
org.springframework.web.HttpLogging      : [49f91090-1, L:/
[0:0:0:0:0:0:0:1]:8080 - R:/[0:0:0:0:0:0:0:1]:50773] Handling completed
```

Thanks to these two properties, we have logged the invoked URL, the request body, and the response. Let us try to invoke a GET request **`http :8080/products/YYYY`**. The log will be the following:

```
o.s.w.s.adapter.HttpWebHandlerAdapter    : [ec8925cd-1] HTTP GET "/prod-
ucts/YYYY", headers={masked}
org.springframework.web.HttpLogging      : [ec8925cd-1] Encod-
ing [class ProductResponse {
    class BaseProduct {
        code: YYYY
        name: zzzz-product
        category: laptop
        price: 1000
        brand: brand1
    }
}]
o.s.w.s.adapter.HttpWebHandlerAdapter    : [ec8925cd-1] Complet-
ed 200 OK, headers={masked}
org.springframework.web.HttpLogging      : [ec8925cd-1, L:/
[0:0:0:0:0:0:0:1]:8080 - R:/[0:0:0:0:0:0:0:1]:50817] Handling completed
```

**Note: The header in the logs is masked by default because it may contain sensitive data. If you want to enable it, add the spring.codec.log-request-details=true property in the `application.properties` file.**

If the body of our request or response contains sensitive data that we cannot log, we can still easily solve this problem by exploiting two concepts:

- In the logs, the Java class that has already been deserialized is printed. We can then exploit the **`toString`** method to exclude any fields to be printed.
- Since the **`toString`** method is autogenerated by the OpenAPI Generator plugin, we need to find a way to have it exclude certain fields from **`toString`**. The OpenAPI specification comes to our rescue, with the password format of the string schema. The description of this format is as follows: *A hint to UIs to obscure input* (**https://**

**spec.openapis.org/oas/v3.1.0#data-types**). It is clear from the description that this is exactly the case here. By adding this format to a field in the OpenAPI file, the plugin will not generate a printout of the latter.

Let us see how it works. In the **`catalog-service.yml`** file, edit the **`name`** field of the BaseProduct schema like this:

```
name:
  type: string
  example: Product Name
  format: password
```

Regenerate the classes with the usual Maven command and start the application. Now try making a GET request, and then, proceed. You will see that the response is logged in the following way:

```
    class BaseProduct {
        code: YYYY
        name: *
        category: laptop
        price: 1000
        brand: brand1
    }
```

The **`name`** field is not printed in plain text, but an asterisk is obscured. Since name is not a true sensitive field, you can reverse the change. Now, you know how to mask a sensitive field in the logs.

# Functional endpoints in Spring WebFlux

Spring WebFlux offers two ways to create HTTP endpoints. The first is the traditional mode, the one we have used so far. This model is the same one that is used in Spring MVC:

```
Mono<ResponseEntity<Void>> addProduct(@RequestBody @Valid
                                      Mono<ProductRequest> productRequest,
                                      final ServerWebExchange exchange);
```

The other way is via functional endpoints, where HTTP requests and responses are treated as functions. Instead of creating a controller class, you need to create a class that acts as a router:

```
@Configuration
public class ProductRouter {
```

```
    @Bean
        public  RouterFunction<ServerResponse>  addProductRouter(ProductHan-
dler productHandler) {
        return route(POST("/products/v2")
                .and(accept(MediaType.APPLICATION_JSON)),
                productHandler::addProduct);
    }
}
```

Router classes simply create endpoints and direct them to the right handlers. Handlers are responsible for handling requests and responses:

```
@Component
@RequiredArgsConstructor
public class ProductHandler {
    private final ProductMapper productMapper;
    private final ProductRepository productRepository;

    public Mono<ServerResponse> addProduct(ServerRequest request) {
        return request.bodyToMono(ProductRequest.class)
                .flatMap(this::insertNewProduct)
               .map(entitySaved -> request.uri() + "/" + entitySaved.code())
                .flatMap(uriString ->
                    ServerResponse.created(URI.create(uriString)).build());
    }
    //...
}
```

There is no one best mode; however, most people prefer the traditional mode because they find the handler classes as a mix between a controller and a service, with a dual responsibility. If you want to learn more about functional endpoints, visit the following link: **https://docs.spring.io/spring-framework/reference/web/webflux-functional.html**.

# Reactive REST client with Spring WebFlux

Spring WebFlux includes a reactive HTTP client called **WebClient**. This client in turn needs to use an HTTP client library. The client libraries supported out-of-box by WebClient are:

- Reactor Netty
- JDK HttpClient
- Jetty reactive HttpClient
- Apache HttpComponents

However, you can also use other libraries by implementing the **`ClientHttpConnector`** interface.

The easiest way to create an instance of the **`WebClient`** class is to use its factory methods:

- **`WebClient.creare()`**
- **`WebClient.create(String baseUrl)`**

In order to customize configurations such as connection or read timeout, you need to instantiate an HTTP client object specific to the library you are using and pass it to WebClient with the **`clientConnector`** method. For example, if you wanted to use the default library, Reactor Netty, you could configure WebClient in this way:

```
var httpClient = HttpClient.create()
        .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000)
        .doOnConnected(conn -> conn
                .addHandlerLast(new ReadTimeoutHandler(10)));


var webClient = WebClient.builder()
        .clientConnector(new ReactorClientHttpConnector(httpClient))
        .build();
```

To make GET requests with WebClient, you need to use the **`retrieve()`** method, which allows you to handle the response from the invoked endpoint. For example, in the following piece of code, you turn the response into a **`ResponseEntity`**:

```
WebClient webClient = WebClient.create("http://localhost:8080");
Mono<ResponseEntity<User>> productResponseMono = webClient.get()
        .uri("users/1")
        .retrieve()
        .toEntity(User.class);
```

If you do not care about the response being embedded in a **`ResponseEntity`** class, you can call the **`bodyToMono`** or **`bodyToFlux`** method instead of the **`toEntity`** method, as shown by the following piece of code:

```
Mono<User> productResponseMono = webClient.get()
        .uri("users/1")
```

```
        .retrieve()
        .bodyToMono(User.class);
```

Although we have shown you how to retrieve a response embedded in the **ResponseEntity** class, you will not be able to write logic if the endpoint receives an HTTP error. In fact, WebClient by default, for all 4xx and 5xx errors, throws an exception of type **WebClientResponseException**. If you want to handle the response based on HTTP statuses, you can use the **onStatus** method, as shown below:

```
Mono<ProductResponse> productResponseMono = webClient.get()
        .uri("products/code1")
        .retrieve()
        .onStatus(HttpStatusCode::is4xxClientError, responseError ->
                responseError.toBodilessEntity()
                        .doOnNext(responseEntity ->
                                log.error("Error: {}", responseEntity))
                        .flatMap(voidResponseEntity ->
                                Mono.error(new RuntimeException())))
        .bodyToMono(ProductResponse.class);
```

As for POST requests, you can insert the request body with the **body()** method:

```
Mono<ResponseEntity<Void>> entityMono = webClient.post()
        .uri("/users")
        .body(userMono, User.class)
        .retrieve()
        .toEntity(Void.class);
```

**PUT** and **DELETE** requests are analogous to POST and GET requests.

Since the catalog service microservice does not currently make HTTP calls to another component, we can see the WebClient class at work on a test method, invoking the catalog service's own API.

# Testing reactive APIs

To test the catalog service API, you can use WebClient and you can mock the **ProductService** class. In the test folder, create the **api** package and write the following class:

```
@SpringBootTest(webEnvironment =
        SpringBootTest.WebEnvironment.DEFINED_PORT)
class ProductControllerTest {
```

```
    @MockBean
    private ProductService productService;

    @Test
    void findProductByCodeOkTest() {
        var webClient = WebClient.create("http://localhost:8080");
        var responseExpected = new ProductResponse()
                .code("code1")
                .category(ProductCategory.LAPTOP)
                .price(1000L);
        when(productService.findProductByCode("code1"))
                .thenReturn(Mono.just(response));

        Mono<ProductResponse> productResponseMono = webClient.get()
                .uri("products/code1")
                .retrieve()
                .bodyToMono(ProductResponse.class);

        StepVerifier.create(productResponseMono)
                .expectNext(responseExpected)
                .verifyComplete();
    }
}
```

The test class starts our application on the real port, which by default is 8080, due to the **`WebEnvironment.DEFINED_PORT`** property. The method verifies that for product code code1, the service responds without error; therefore, the endpoint must return HTTP status code 200 with the response body. We verify this through the **`StepVerifier`** class, which allows us to consume a responsive stream and make assertions.

Spring WebFlux provides a WebClient type suitable for testing, called **`WebTestClient`**. By using this class, you can avoid setting the server URL and you can avoid using the **`StepVerifier`** class, since **`WebTestClient`** already has methods for performing tests on the response. We can then use **`WebTestClient`** in our test class:

```
@WebFluxTest(ProductController.class)
class ProductControllerTest {
```

```
    @MockBean
    private ProductService productService;

    @Autowired
    private WebTestClient webTestClient;

    @Test
    void findProductByCodeOkTest() {
        var responseExpected = new ProductResponse()
                .code("code1")
                .category(ProductCategory.LAPTOP)
                .price(1000L);
        when(productService.findProductByCode("code1"))
                .thenReturn(Mono.just(responseExpected));

        webTestClient.get()
                .uri("/products/code1")
                .exchange()
                .expectBody(ProductResponse.class)
                .consumeWith(result -> assertThat(result.getResponseBody())
                        .isEqualTo(responseExpected));

    }
}
```

We annotated the class with **`@WebFluxTest`**, which, unlike **`@SpringBootTest`**, does not create all of Spring's context, only the context to handle controllers. By specifying the controller in the annotation, Spring creates the context for testing the specific controller. In addition, this annotation allows you to inject an instance of **`WebTestClient`**. The new test method performs the same tests as before but in a more elegant way.

Let us now see how to test the POST API **`addProduct`**:

```
@Test
void addProductOk() {
    var request = new ProductRequest()
```

```
            .code("code1")
            .category(ProductCategory.LAPTOP)
            .price(1000L);
   when(productService.addProduct(request)).thenReturn(Mono.just("code1"));
    webTestClient.post()
            .uri("/products")
            .bodyValue(request)
            .exchange()
            .expectStatus().isCreated()
            .expectBody(Void.class)
            .consumeWith(result ->
                    assertThat(result.getResponseHeaders().get("Location").
get(0))
                            .isEqualTo("/products/code1"));
}
```

The method above tests that after giving a valid request, the API returns status code 201 with the header **`Location`** value.

**Note: In the test, the header value is a relative URL, not absolute, since using WebTestClient, the WEB environment is mocked. In the GitHub repository, you will find all the tests for KO cases and on the PUT and DELETE endpoints.**

# Spring MVC vs. Spring WebFlux performance

For applications with high concurrency demands, choosing between traditional synchronous models, such as Servlet-based Spring MVC, and asynchronous, non-blocking frameworks like WebFlux is critical. To evaluate how WebFlux performs compared to a standard Spring MVC Servlet application, we conducted a load test using Apache Benchmark (**https://httpd.apache.org/docs/2.4/programs/ab.html)**, focusing on total test duration, throughput, average request handling time, and data transfer rates.

We used a simple Spring Boot application that provides a REST endpoint that returns a list of ten books. This is the application code in the version with Spring MVC:

```
@SpringBootApplication
public class VrApplication {

      @RestController
```

```
	public class MockApi {

		@GetMapping
		public ResponseEntity<List<Book>> findAll()
					throws InterruptedException {
			var books = new ArrayList<Book>();
			Thread.sleep(500L);
			for(int i = 0; i < 10; i++) {
				books.add(new Book(i, UUID.randomUUID().to-
String()));
			}
			return ResponseEntity.ok(books);
		}
	}

	record Book(int bookId, String bookName){}

}
```

This is the code for the version with Spring WebFlux:

```
@SpringBootApplication
public class WebfluxApplication {

	public static void main(String[] args) {
		SpringApplication.run(WebfluxApplication.class, args);
	}

	@RestController
	public class MockApi {

		@GetMapping
		public Mono<ResponseEntity<Flux<Book>>> findAll() {
			Flux<Book> booksFlux = Flux.range(0, 10)
					.delayElements(Duration.ofMillis(50))
					.map(i -> new Book(i,
						UUID.randomUUID().to-
String()));
```

```
                return Mono.just(ResponseEntity.ok(booksFlux));
            }

        }

        record Book(int bookId, String bookName){}
}
```

The load test involved 10000 total requests with a competition factor of 250. The applications were released as a JAR package in an Amazon EC2 machine t3.micro (2 vCPUs and 1 GiB of memory).

The test results are summarized in the *Table 2.2*:

| Metric | Spring MVC | Spring WebFlux | Better Value |
|---|---|---|---|
| **Time taken for tests (s)** | 26.538 | 23.007 | Lower is better |
| **Requests per second (#/sec)** | 376.82 | 434.66 | Higher is better |
| **Time per request (ms, mean)** | 663.44 | 575.163 | Lower is better |
| **Transfer rate (Kbytes/sec)** | 270.84 | 339.15 | Higher is better |

***Table 2.2:*** *Results of load test between Spring MVC and Spring WebFlux*

The benchmark results underscore significant performance improvements with WebFlux. The WebFlux application completed 10,000 requests in 23.007 seconds, notably faster than the traditional Servlet application, which took 26.538 seconds. This reduction in total test duration highlights WebFlux's capability to handle concurrent requests more efficiently under heavy load conditions.

In terms of throughput, WebFlux achieved 434.66 requests per second, a substantial increase over the 376.82 requests per second achieved by the Servlet application. This higher throughput indicates WebFlux's efficiency in managing concurrent connections, supporting a greater request rate with minimal impact on response times. Such performance gains make WebFlux particularly advantageous for scalable, high-traffic environments.

When examining the average time per request, WebFlux demonstrated an improved mean time of 575.163 ms compared to the Servlet application's 663.440 ms. This suggests that WebFlux's non-blocking architecture can reduce per-request processing times, likely due to its ability to manage concurrency without the overhead of a traditional thread-per-request model. The decreased mean request time highlights how WebFlux's reactive design can optimize application responsiveness by allowing the system to process more requests concurrently.

In terms of data transfer, WebFlux recorded a transfer rate of 339.15 Kbytes/sec, outperforming the Servlet application's rate of 270.84 Kbytes/sec. This higher data transfer rate indicates that WebFlux handles data exchanges efficiently,

tensive applications where throughput is critical for performance. This advantage implies that WebFlux may be better suited for applications requiring high data transfer rates alongside scalable concurrency handling.

This comparison illustrates that WebFlux offers substantial performance benefits over a traditional Servlet-based Spring MVC application. With a faster total test duration, improved throughput, reduced time per request, and enhanced data transfer rate, WebFlux provides a robust, scalable solution that excels in handling concurrent requests. For applications facing high concurrency demands, the reactive, non-blocking model of WebFlux is an excellent choice, enabling improved resource utilization and responsiveness. However, for applications where synchronous processing is essential or reactive programming is not a core requirement, the Servlet model remains a viable approach.

# Conclusion

In this chapter, we showed you how to write code with reactive programming in Java. With the reactive paradigm, we coded responsive REST APIs with Spring WebFlux by using the API-first approach with the OpenAPI Generator plugin. The API-first approach allows you to focus first on the API to be written and only after it is fully drafted do you write the code. You also saw how to centrally handle errors with WebFlux without dirtying the business logic. Finally, we showed you how to test APIs with the testing framework provided by Spring. In the next chapter, we will show you how to write APIs with Spring MVC by using virtual threads.

# Points to remember

- Spring WebFlux used the event loop approach.
- Spring WebFlux implements the Reactive Streams specification via the Reactor library.
- In reactive programming, the publisher does not publish events until it is subscribed to by a subscriber.
- Use @WebFluxTest to test REST APIs with Spring WebFlux.

# Exercises

1. Try implementing the PUT and DELETE endpoints.
2. Try implementing the GET paged endpoint to retrieve all products.
3. Try implementing tests for the other endpoints.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

**https://discord.bpbonline.com**



# Chapter 3
# Easily Scalable APIs with Virtual Threads

## Introduction

In this chapter, we will discover Java's virtual threads to explore how they revolutionize API scalability. Virtual threads, a groundbreaking feature in Java, offer a lightweight and efficient alternative to traditional threads, making them an ideal choice for building highly scalable applications. In this chapter, we embark on a comprehensive journey, beginning with an in-depth understanding of virtual threads in Java. Armed with this knowledge, we then transition to practical implementation, guiding you through the process of setting up a Spring MVC project utilizing virtual threads to architect robust and scalable APIs. Furthermore, we introduce RestClient, a modern synchronous REST client tailored to harness the full potential of virtual threads, offering enhanced performance and scalability. As we progress, we will explore testing methodologies within the Spring MVC framework, ensuring your APIs meet the highest standards of reliability and functionality. Finally, we engage in a comparative analysis, contrasting virtual threads with WebFlux to elucidate the optimal choice for your specific use cases.

## Structure

In this chapter, we will discuss the following topics:

- Understanding virtual threads in Java
- Getting Started with Spring MVC with virtual threads

- A new modern synchronous REST client: RestClient
- Testing in Spring MVC
- Spring MVC performance comparison: Platform vs virtual threads
- Virtual threads vs. WebFlux: what to choose?

# Objectives

By the end of the chapter, you will have a deep understanding of virtual threads and their critical role in creating resilient and scalable APIs. You will know in detail what is behind virtual thread technology, such as linking with carrier threads that allow you to have an association, not 1:1, with operating system threads, and you will learn about the continuation technique used by virtual threads to preserve their state when the **Java Virtual Machine** (**JVM**) suspends them. You will have a thorough understanding of Spring MVC, the annotations it uses, the patterns it uses, and error handling. You will also learn how to create paginated APIs with Spring Data. By the end of the chapter, we will have all the knowledge, so that you can decide whether to adopt the latter or Spring WebFlux for your applications.

# Understanding virtual threads in Java

Virtual threads are a new killer feature of Java, introduced in preview mode with JDK 19 and incorporated permanently in **Java Development Kit** (**JDK**) 21. Virtual threads are lightweight threads that reduce the effort of writing, monitoring, and debugging concurrent high-throughput applications. Before going into detail about how they work, it behooves us to explain why the concept of threads is so important in Java.

## Platform threads vs. virtual threads

Java is a multi-threaded language. Each instruction is executed on a thread, and because of the multi-threading, multiple instructions can be executed in parallel. Each thread provides a stack to store local variables and a context when an exception is thrown. Developers can read the thread's stack trace to analyze an error. Before the virtual threads feature, the concurrency model in Java was uniquely implemented through platform threads.

**Platform threads** are wrappers of operating system threads. We can say that platform threads are associated 1:1 with operating system threads. Creating many threads with this concurrency model, therefore, involves consuming a lot of computational resources. The number of threads you can create depends on the number of threads supported by the operating system.

**Virtual threads**, like platform threads, are instances of **`java.lang.Thread`**, but they are not associated 1:1 with operating system threads. Specifically, when a virtual thread is created,

it is associated with an operating system thread, but if it encounters blocking operations such as I/O operations, the JVM suspends it until the blocking operation is finished. In the meantime, the OS thread is free to be associated with another virtual thread.

# Virtual threads to simplify scalability in Java

In the previous chapter, we explained that the traditional model for handling HTTP requests is the thread-per-request. Server applications to handle concurrent requests associate a thread with each HTTP request. The thread is released only when the request processing is completed. This programming style is easy to understand, implement, and debug. However, with platform threads, there is limited thread availability because they are wrappers of the operating system threads. OS threads are expensive, in terms of creation and management. It is not even enough to use the **thread pool** technique of grouping an already defined number of threads to eliminate the cost of starting new threads. If each HTTP request consumes an operating system thread, for its entire duration, the number of threads often becomes the limiting factor long before other resources, such as CPU or network connection.

The technique of asynchronous programming, particularly the reactive paradigm, can be used to overcome this problem. However, you saw in the previous chapter that reactive programming is much more complex than imperative programming, both in terms of writing code and debugging.

Virtual threads allow you to have, through efficient thread management, highly scalable applications written using the traditional (and simple) thread-per-request technique. The result is that you can have the same scalability as asynchronous programming, but in a transparent way, facilitating software development.

# Virtual threads in detail

Virtual threads in Java came through **Project Loom**. They were initially called **fibers**. Virtual threads, as opposed to platform threads, are scheduled not by the operating system, but by the JVM. This makes it possible to overlook the time involved in thread creation and **context switching**, which is the technique used by the operating system to switch from one process to another on a single CPU. Of course, there is still an association with OS threads, but it is not 1:1 as with platform threads. Specifically, the JVM associates a virtual thread with a platform thread called a **carrier thread**. The virtual thread remains associated with the carrier thread until it encounters a blocking operation. When the virtual thread encounters a blocking operation, the JVM suspends it so it can associate the carrier thread with another virtual thread. *Figure 3.1* shows the association between virtual threads, carrier threads, and OS threads:

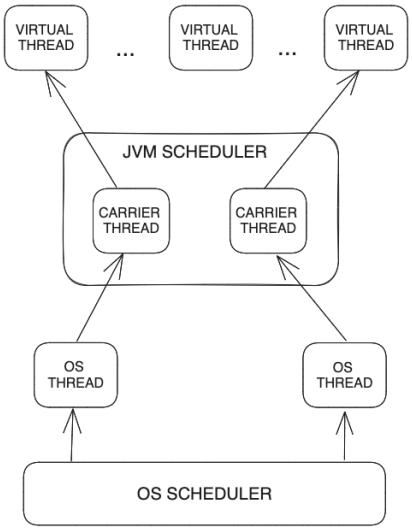



***Figure 3.1**: Virtual threads scheduling*

Suspending virtual threads works with the Continuations technique. In general, this technique allows you to pause the execution of a program by saving its state, and then resume execution exactly from the point where it was suspended.

Creating a virtual thread also involves creating an object of type **`jdk.internal.vm.Continuation`** that contains the current state of the thread, such as the call stack and local variables. When the virtual thread needs to perform a blocking operation, it is suspended by the JVM, and the Continuation object is saved in heap memory. Meanwhile, the carrier thread can be associated with another virtual thread. When the blocking operation is finished, the virtual thread is restored due to the Continuation object and can resume its execution on another carrier thread. The virtual thread stack compared to the platform thread stack occupies little memory (order of megabytes versus hundreds of bytes) and the stacks are stored in the JVM heap rather than in memory allocated by the operating system. Because of this, virtual threads are cheaper, and thus an application can use hundreds of miles of virtual threads.

Let us look at an example of virtual thread execution and platform threads. We create a class that implements the Runnable interface:

```
class Task implements Runnable {
    private final int taskNumber;

    public Task(int taskNumber) {
        this.taskNumber = taskNumber;
    }
```

```
    @Override
    public void run() {
        if (taskNumber == 1) {
            System.out.println(Thread.currentThread());
        }
        try {
            Thread.sleep(Duration.ofMillis(10));
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        if (taskNumber == 1) {
            System.out.println(Thread.currentThread());
        }
    }
}
```

This class has a field of type integer called **`taskNumber`**. In the implementation of the **`run()`** method, if this field is equal to 1, the current thread is printed, then the current thread is paused for ten milliseconds, and finally, it is reprinted. We use this class with platform threads:

```
@Test
void platformThread() throws InterruptedException {
    var platformThread = IntStream.rangeClosed(1, 10)
            .mapToObj(taskNumber ->
                    Thread.ofPlatform().unstarted(new Task(taskNumber))).
toList();
    platformThread.forEach(Thread::start);
    for (Thread t : platformThread) {
        t.join();
    }
}
```

This method creates ten platform threads using the static **`Thread.ofPlatform()`** method. Each thread is associated with an object of type **`Task`**. By executing the method, we would get a printout of this type:

```
Thread[#22,Thread-0,5,main]
Thread[#22,Thread-0,5,main]
```

This printout shows that the thread associated with **`taskNumber`** 1 is always the same, both before and after it is paused.

We use the same method by creating virtual threads this time:

```
@Test
void virtualThread() throws InterruptedException {
    var virtualThreads = IntStream.rangeClosed(1, 10)
            .mapToObj(taskNumber ->
                    Thread.ofVirtual().unstarted(new Task(taskNumber))).
toList();
    virtualThreads.forEach(Thread::start);
    for (Thread t : virtualThreads) {
        t.join();
    }
}
```

This time we used the **`Thread.ofVirtual()`** method to create the virtual threads. Running the code, we would get a printout like this:

```
VirtualThread[#22]/runnable@ForkJoinPool-1-worker-1
VirtualThread[#22]/runnable@ForkJoinPool-1-worker-9
```

In this printout, you can see that the virtual thread with id 22, is associated with two different platform threads (the carrier threads). This is because the virtual thread has encountered the **`Thread.sleep()`** blocking operation. From the printout, you can also see that the virtual threads use by default a scheduler of type **`ForkJoinPool`**, which differs from the other **`ExcutorServices`** in that it uses **work-stealing**.

Work-stealing is a scheduling algorithm used to optimize CPU resource utilization and mitigate possible congestion points. The **`ForkJoinPool`** class uses a scheduling model based on the concept of *tasks*, where a task represents a unit of work that can be executed in parallel with others. The thread pool of the **`ForkJoinPool`** consists of a set of worker threads, each of which is associated with a work queue containing the tasks to be executed. The work-stealing algorithm allows thread workers to *steal* tasks from the work queue of other thread workers when their queue is empty. This mechanism promotes a dynamic distribution of workload among thread workers and helps maintain an even utilization of CPU resources, avoiding situations where some threads are overloaded while others are idle.

Virtual threads are suitable for applications that have many blocking operations. They are not suitable for CPU-intensive applications. Also, with the fact that you can create thousands of threads, it may store too much data in context (think about the use of **`ThreadLocal`**).

With virtual threads, Java version 21 introduced a killer feature that completely changed the concurrency model. In the next section, you will see that the use of virtual threads in Spring MVC is completely transparent.

# Getting started with Spring MVC with virtual threads

Spring MVC is the most widely used Java framework for creating REST services because of its ease of use, which is based on the concept of *Open for extension, closed for modification*. In this section, we will see how to create the order service microservice, responsible for order management, with Spring MVC, using virtual threads. In addition, this microservice will contact the catalog service microservice to retrieve some product data from the orders. All the code in this chapter is available in the **chapter-03** folder of the GitHub repository.

# Introduction to Spring MVC

The Spring MVC framework is called this way because it allows the **Model-View-Controller** pattern to be implemented. In the past, before the development of libraries and frameworks such as React and Angular, web pages were generated by the server, not the client. The Model-View-Controller pattern divides the logic of data management, manipulation, and visualization into components. Specifically, the model component is responsible for managing and manipulating the business model, the view component is responsible for visualizing the model data, and the controller component is responsible for passing model data to the view and vice versa. Although this pattern is still used today, the Spring MVC module is widely used much more for creating REST services.

Spring MVC follows the thread-per-request model and is designed around a Servlet that handles requests and dispatches them to controllers, implementing the **Front Controller** pattern. Specifically, the Servlet that acts as the Front Controller in Spring is called the **`DispatcherServlet`**, which receives HTTP requests, and via a **`HandlerMapping`**, which is responsible for mapping URLs to Controller classes (annotated with **`@Controller`**) and its methods (annotated with **`@RequestMapping`**), invokes the Controller class method mapped to the input endpoint. *Figure 3.2* shows what has been described so far:

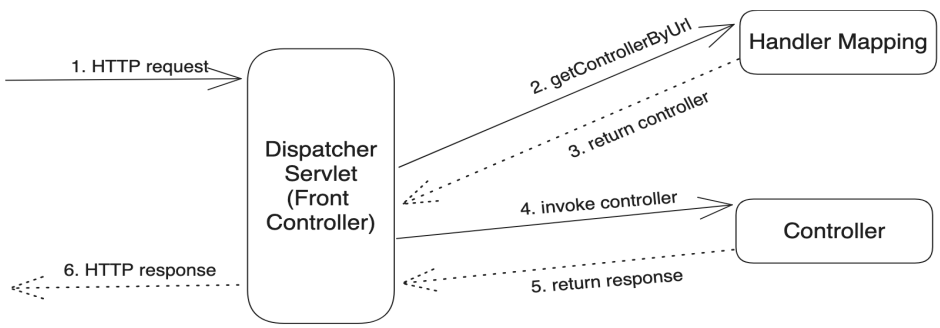



*Figure 3.2*

Spring MVC with Spring Boot uses Tomcat by default as the Servlet Container, but you can easily switch Servlets by excluding the Tomcat dependency and inserting that of another Servlet Container, such as Jetty.

Although the API-first approach hides the use of Spring MVC annotation, you should know which ones are most important, refer to the following table:

| Annotation | Description |
|---|---|
| `@RestController` | Annotation that groups `@Controller` and `@ResponseBody`. It tells Spring that the class is a Controller, and the response is an HTTP response, not the name of a view. |
| `@RequestMapping` | Annotation that can be put at the class or method level to indicate the path to a URL. |
| `@PathVariable` | Associated with a parameter of a Controller's method, it indicates that the parameter is a path parameter. |
| `@RequestParam` | Associated with a parameter of a Controller's method, it indicates that the parameter is a query parameter. |
| `@RequestBody` | Associated with a parameter of a Controller's method, it indicates that the parameter is an HTTP request body. |
| `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, `@PatchMapping` | It is composed of annotations that act as a shortcut for `@RequestMapping(method = RequestMethod.GET)`, `@RequestMapping(method = RequestMethod.POST)`, etc.<br>• `@GetMapping`: annotation for mapping GET requests. Useful when you want to retrieve data from a resource.<br>• `@PostMapping`: annotation for mapping POST requests. Useful when we want to save a new resource.<br>• `@PutMapping`: annotation for mapping PUT requests. Useful when we want to overwrite a resource with new data.<br>• `@DeleteMapping`: annotation for mapping DELETE requests. Useful when we want to remove a resource.<br>• `@PatchMapping`: annotation for mapping PATCH requests. Useful when we want to make partial changes to a resource. Compared to `@PutMapping`, only the fields in the request are updated. |

***Table 3.1:*** *The most commonly used annotations in the Spring MVC module*

**Note: These annotations are also used in the Spring WebFlux module.**

# Spring MVC in practice

In order to create the project from scratch, we will rely as usual on Spring Initializr. The modules to be imported are the same as in the last chapter, except we replace the two asynchronous modules with synchronous ones, that is, we replace the WebFlux module with **Web** and the Data R2DBC module with **Data JDBC**. *Figure 3.3* shows the Spring Initializr screen for creating the new microservice:

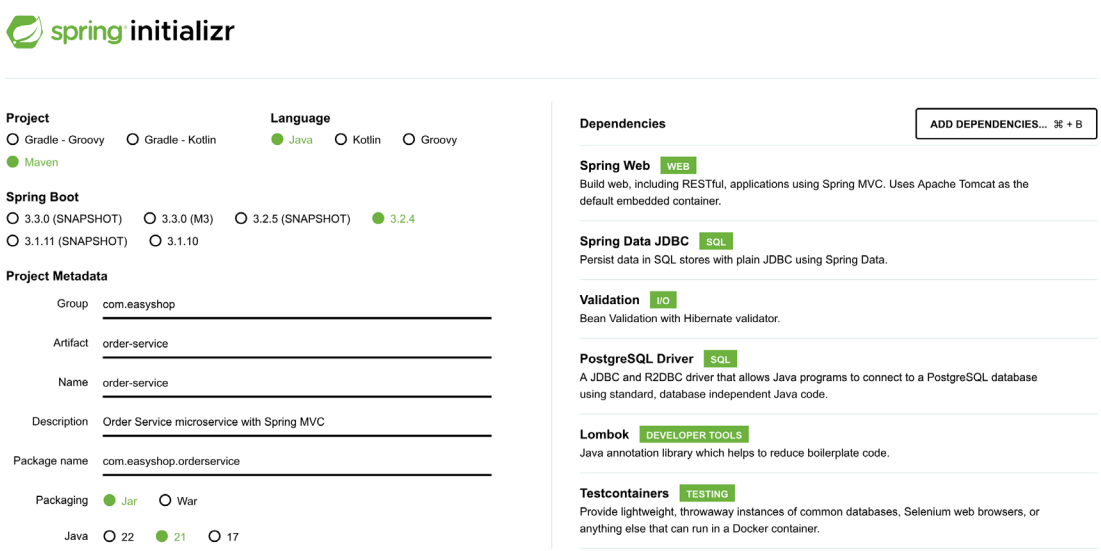



***Figure 3.3**: The setup for a project with Spring MVC*

The Spring Web module allows importing the dependency of Spring MVC. The Spring Data JDBC module allows importing the dependency of the module of the same name. This module uses the concept of **aggregate** and **aggregate root** of **Domain Driven Design**. An aggregate is a cluster of domain objects that can be treated as a single unit and are associated with a single aggregate root. Any operation from the outside must be done only on the aggregate root. The root can thus guarantee the integrity of the aggregates. A good example of this is an order entity that acts as the aggregate root to the order items entities, which are part of the aggregate.

We will also use MapStruct and Springdoc dependencies for this microservice.

**Note: The Springdoc dependency is needed, as mentioned in the previous chapter, the OpenAPI Generator plugin, since it uses Swagger annotations. However, since we are using Spring MVC and not WebFlux, the dependency to be imported is not springdoc-openapi-starter-webflux-api but springdoc-openapi-starter-webmvc-api.**

As for the business model, we might initially think of the order object consisting of the following fields:

- A unique order code.
- A status, which could be taken charge, rejected, or delivered.
- A creation date and an update date of the order.
- A list of product codes from the order. Each product code is associated with a quantity.
- The total price of the order.

This domain entity could be mapped to the database in the following way:

```
CREATE TABLE IF NOT EXISTS orders (
    order_id SERIAL PRIMARY KEY,
    order_code VARCHAR UNIQUE NOT NULL,
    status VARCHAR,
    total_price BIGINT,
    created_at TIMESTAMPTZ,
    updated_at TIMESTAMPTZ
);
CREATE TABLE IF NOT EXISTS order_items (
    order_id INTEGER,
    product_code VARCHAR,
    quantity INTEGER,
    PRIMARY KEY(order_id, product_code),
    CONSTRAINT fk_purchase_orders
        FOREIGN KEY(order_id) REFERENCES orders(order_id)
);
```

The first table, **`orders`**, is responsible for mapping the summary data for an order, while the **`order_items`** table is responsible for associating one or more products with an order. You will find the file **`init.sql`** updated with these two tables.

Let us design the API for creating an order:

```
/orders:
  post:
    tags:
      - order
    summary: Add a new order
    operationId: addOrder
```

```
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/OrderRequest'
      required: true
    responses:
      '201':
        description: Successful operation
        headers:
          Location:
            schema:
              type: string
              format: uri
      '400':
        description: Invalid input
        content:
          application/problem+json:
            schema:
              $ref: '#/components/schemas/ProblemDetail'
```

Against a valid request body, the API returns the HTTP 201 status with the location header containing the URI of the created resource. The request body is as follows:

```
OrderRequest:
  required:
    - products
  type: object
  properties:
    products:
      type: array
      items:
        $ref: '#/components/schemas/ProductOrder'
ProductOrder:
  required:
    - productCode
```

```
    - quantity
  type: object
  properties:
    productCode:
      type: string
    quantity:
      type: integer
```

It contains an array of **`productCode`** and **`quantity`** pairs. Find the complete OpenAPI in the file: **`order-service/openapi/server/order-service.yml`**.

The sequence diagram of this API is shown in *Figure 3.4*:

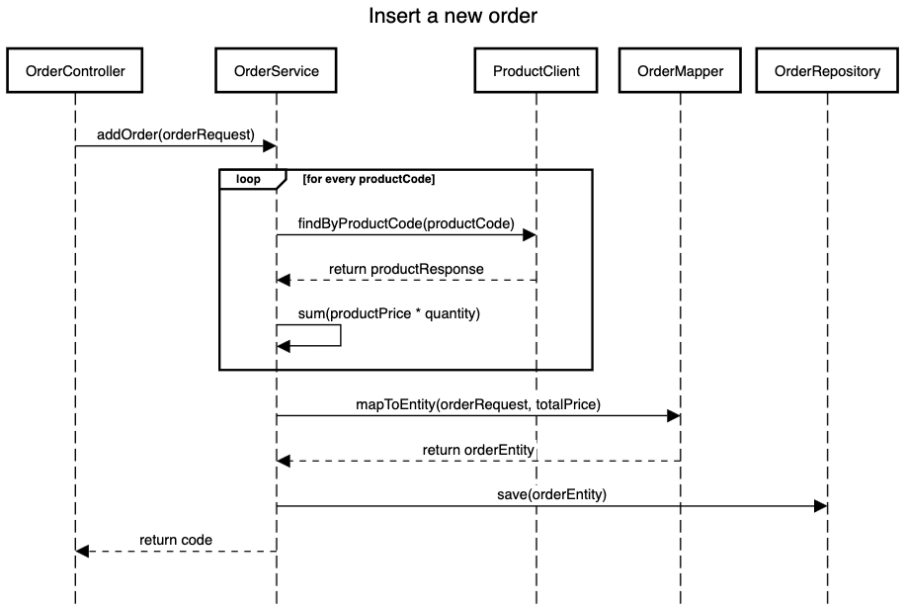



***Figure 3.4**: Sequence diagram of the operation of inserting a new order*

The diagram is like inserting a new product operation, it can be seen in the previous chapter. In this case, however, there is more complexity since the operation also involves, via the **`ProductClient`** class, requesting product information for each **`productCode`** received as input. Specifically, **`ProductClient`** makes an HTTP call to the **`findProductByCode`** endpoint of the catalog service microservice.

Add the OpenAPI Generator plugin in the **`pom.xml`** file to autogenerate API models and interfaces. Refer to the following code for a better understanding:

```
<plugin>
    <groupId>org.openapitools</groupId>
```

```
    <artifactId>openapi-generator-maven-plugin</artifactId>
    <version>7.4.0</version>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
            <configuration>
                <inputSpec>
                    ${project.basedir}/openapi/server/order-service.yml
                </inputSpec>
                <openapiNormalizer>REF_AS_PARENT_IN_ALLOF=true</openapiNormalizer>
                <generatorName>spring</generatorName>
                <apiPackage>com.easyshop.orderservice.generated.api</apiPackage>
                <modelPackage>
                    com.easyshop.orderservice.generated.model
                </modelPackage>
                <schemaMappings>
                    ProblemDetail=org.springframework.http.ProblemDetail
                </schemaMappings>
                <typeMappings>
                    OffsetDateTime=java.time.Instant
                </typeMappings>
                <configOptions>
                    <interfaceOnly>true</interfaceOnly>
                    <delegatePattern>true</delegatePattern>
                    <useSpringBoot3>true</useSpringBoot3>
                    <openApiNullable>false</openApiNullable>
                    <useTags>true</useTags>
                </configOptions>
            </configuration>
        </execution>
    </executions>
</plugin>
```

The configuration properties of the OpenAPI Generator plugin are similar to those in the previous chapter, except that the **`<reactive>true</reactive>`** tag has not been added because we want to use Spring MVC and not WebFlux. From the project root, we run the **`./mvnw clean compile`** command to autogenerate the classes with OpenAPI Generator plugin. Let us look at the OrderAPI autogenerated interface found inside the **`chapter-03/order-service/target/generated-sources/openapi/src/main/java/com/easyshop/orderservice/generated/api`** folder:

```
public interface OrderApi {

    //...
    @RequestMapping(
            method = RequestMethod.POST,
            value = "/orders",
            produces = {"application/problem+json"},
            consumes = {"application/json"}
    )

    default ResponseEntity<Void> _addOrder(
            @
Parameter(name = "OrderRequest", description = "", required = true)
            @Valid @RequestBody OrderRequest orderRequest
    ) {
        return addOrder(orderRequest);
    }

    // Override this method
    default ResponseEntity<Void> addOrder(OrderRequest orderRequest) {
        //...
    }
}
```

**Note: Compared to the previous chapter, the ResponseEntity class is not wrapped in a Mono because we are not in a reactive context.**

Let us now implement the classes designed in the sequence diagram. We start with the **`OrderEntity`** class that maps to the order table. Create the **`middleware.db.entity`** package and write the following Java record:

```
@Table("orders")
@Builder(toBuilder = true)
public record OrderEntity(
        @Id
        Long orderId,
        String orderCode,
        String status,
        @MappedCollection(idColumn = "order_id")
        Set<OrderItem> items,
        Long totalPrice,
        @CreatedDate
        Instant createdAt,
        @LastModifiedDate
        Instant updatedAt
) {}
```

Compared with **`ProductEntity`**, this record contains the following new annotations:

- The **`@MappedCollection`** annotation allows us to specify the list of objects on which the one-to-many relation is realized. With the **`idColumn`** parameter, we specify the foreign key name of the child table.
- The **`@CreatedDate`** and **`@LastModifiedDate`** annotations allow the **`createdAt`** and **`updatedAt`** fields to be automatically enhanced with the creation and modification dates, respectively.

In the same package, write the **`OrderItem`** record:

```
@Table("order_items")
public record OrderItem(
        String productCode,
        int quantity
) {}
```

The **`@CreatedDate`** and **`@LastModifiedDate`** annotations only work if JDBC auditing is enabled with the **`@EnableJdbcAuditing`** annotation. Create the **`config`** package and write the following class:

```
@Configuration
@EnableJdbcAuditing
public class DataConfig {
}
```

Now, create the **repo** subpackage inside **middleware.db** and write the following repository interface:

```
public interface OrderRepository
        extends CrudRepository<OrderEntity, String>, {
}
```

As with **ProductRepository**, this interface extends **CrudRepository**, although in the previous chapter we used the Spring Data R2DBC module, whereas now we have used the Spring Data JDBC module. This is one of the qualities of Spring Data that we mentioned in the previous chapter, which is the ability to hide the implementation of the imported module.

Write the **ProductClient** interface, which is responsible for providing the data for a product, given an input product code. Create the **msclient** subpackage within *middleware* and write the following interface:

```
public interface ProductClient {
    ProductResponse findByProductCode(String productCode);
}
```

The **Data Transfer Object (DTO**) class, **ProductResponse**, maps the response body of the catalog service. Create the **dto** subpackage within **middleware.msclient** and write the following record:

```
public record ProductResponse(
        String code,
        String category,
        Long price
) {}
```

For now, let us create a mock class that simulates the operation of the client. In the section on **RestClient**, we will create an implementation that makes the HTTP call to the catalog service microservice. Create the **impl** subpackage inside **middleware.msclient** and write the following class:

```
@Component
public class MockProductClient implements ProductClient {
    @Override
    public ProductResponse findByProductCode(String productCode) {
        return new ProductResponse(productCode, null, 10000L);
    }
}
```

Write the mapper interface with a method that, given an **`OrderRequest`** object, the total price of the products in the order, and the order status, creates an object of type **`OrderEntity`**. Create the **`mapper`** package and write the following interface:

```
@Mapper(componentModel = "spring")
public interface OrderMapper {


    @Mapping(target = "orderCode", expression = "java(this.
generateOrderCode())")
    @Mapping(target = "items", source = "orderRequest.products")
    OrderEntity toEntity(OrderRequest orderRequest, long totalPrice,
                         OrderStatus status);


    default String generateOrderCode() {
        var randomString = RandomStringUtils.randomAlphanumeric(5).
toUpperCase();
        return randomString + System.currentTimeMillis();
    }
}
```

Compared to the previous chapter, this interface contains MapStruct configurations that are worth analyzing:

- MapStruct by default tries to map the fields of the target class to fields that have the same name as the source class. When the fields have different names, you can use the **`@Mapper`** annotation to specify the mapping of the fields.
- If a target class field is not to be mapped from a source class field but from a method result, you can use the property **expression**. The **`orderCode`** field of the target class is valued by calling the **`generateOrderCode`** method, which creates a random string.

Now, let us write the class that handles the business logic, **`OrderService`**. Create the **`service`** package and write the following class:

```
@Service
@RequiredArgsConstructor
@Slf4j
@Transactional(readOnly = true)
public class OrderService {
```

```
    private final OrderRepository orderRepository;
    private final ProductClient productClient;
    private final OrderMapper orderMapper;

    @Transactional
    public String addOrder(OrderRequest orderRequest) {
        var totalPrice = calculateTotalPrice(orderRequest);
        var entity = orderMapper.toEntity(orderRequest, totalPrice,
                OrderStatus.TAKEN_CHARGE);
        var orderInserted = orderRepository.save(entity);
        log.info("New order inserted: {}", orderInserted);
        return entity.orderCode();
    }

    private Long calculateTotalPrice(OrderRequest orderRequest) {
        return orderRequest.getProducts().stream()
                .map(this::createQuantityCodePair)
                .map(quantityAndPrice ->
                        quantityAndPrice.getFirst() * quantityAndPrice.
getSecond())
                .reduce(0L, Long::sum);
    }

    private Pair<Integer, Long> createQuantityCodePair(ProductOrder produc-
tOrder) {
        var response = productClient
                .findByProductCode(productOrder.getProductCode());
        return Pair.of(productOrder.getQuantity(), response.price());
    }
}
```

The **addOrder** method does exactly what the sequence diagram describes: for each object of type **ProductOrder**, the **findByProductCode** method of the **ProductClient** class is executed to retrieve the price of each product. The price is multiplied by the quantity of the product. The variable **totalPrice** is the summation of this calculation for all products. The entity of type **OrderEntity** is then created, which is saved in the database, and finally, the order code is returned to the caller.

In the last step, we create the **`OrderController`** class that implements the **`OrderApi`** autogenerated interface and simply calls the service in its method. Create the **`api`** package and write the following class:

```
@RestController
@RequiredArgsConstructor
public class OrderController implements OrderApi {

    private final OrderService orderService;

    @Override
    public ResponseEntity<Void> addOrder(OrderRequest orderRequest) {
        final String orderCode = orderService.addOrder(orderRequest);
        var uri = ServletUriComponentsBuilder
                .fromCurrentRequestUri()
                .toUriString() + "/" + orderCode;
        return ResponseEntity.created(URI.create(uri)).build();
    }
}
```

The **`ServletUriComponentsBuilder`** class allows retrieving useful information such as the URI from the current request, wrapped with the **`HttpServletRequest`** class.

If we started the application as it is now, Spring MVC would use platform threads. To use virtual threads simply set the following property in the **`application.properties`** file: **`spring.threads.virtual.enabled=true`**. All the properties in the **`application.properties`** file are as follows:

```
spring.application.name=order-service
server.port=8081
spring.threads.virtual.enabled=true
spring.datasource.url=jdbc:postgresql://localhost:5432/easyshopdb_order
spring.datasource.username=user
spring.datasource.password=password
```

Let us try to invoke the API. Start Docker and open a terminal at the path **`chapter-03/infra/docker`**, after which first run the command to delete the Postgres volume: **`docker volume rm docker_easyshop-postgres`** and then start the containers with the command: **`docker compose up -d`**. With the IDE, start the main method of the **`OrderServiceApplication`** class. From the terminal, try making the following POST request:

```
http :8081/orders \
  Content-Type:application/json \
  "products[0][productCode]"=SMA-XCV \
  "products[0][quantity]":=1
```

The response will be like this:

```
HTTP/1.1 201
Location: http://localhost:8081/orders/C04RK1711909602708
```

You can verify on pgadmin that the records were correctly saved in the **`orders`** and **`order_items`** tables.

However, because we used a **`mockata`** class to implement **`ProductClient`**, it is currently possible to create orders with products that do not exist in the catalog service database.

Now, try making a POST request omitting a required field, such as **`productCode`**:

```
http :8081/orders \
  Content-Type:application/json \
  "products[0][quantity]":=1
```

You will see a response like this:

```
HTTP/1.1 400
{
    "error": "Bad Request",
    "path": "/orders",
    "status": 400,
    "timestamp": "2024-03-31T19:39:51.812+00:00"
}
```

While correctly returning status code 400, the response body does not meet the **`ProblemDetail`** specification. We will show you how to fix this behavior in the error handling subsection in Spring MVC. You will also find GET (**`findOrderByCode`**), PUT, and DELETE endpoints in the *chapter-03* folder of the GitHub repository.

# Paginated API in Spring MVC

In the first chapter, talking about REST, for APIs that return a list of resources, we recommended that they be designed to provide for pagination. It allows both to not burden the response body, but also to optimize the queries to be made on the database. Pagination usually involves the use of these input parameters:

- A parameter indicating the page number you want to display.
- A parameter indicating the size of the page in terms of elements.
- An optional parameter on sorting, indicating the fields on which to sort the elements on the page, in ascending or descending order.

In response, a page, in addition to returning resource elements, should provide the client with useful information such as the total number of pages, so that the client knows whether a next page exists.

Spring Data provides classes to handle pagination. On the input side, Spring Data provides the **Pageable** interface, which allows the input parameters for pagination listed above to be enhanced. As for output, Spring Data provides the **Page** interface, which incorporates the page's resource list and other information such as the total number of pages.

We design a paged API to retrieve orders. In addition to providing the necessary parameters for pagination, the API will also have to accept an optional parameter to filter items by status. The filter on status is of type **`startWith`**, so if the client values the status parameter with **`DEL`**, orders with the status DELIVERED will be found. We add the following endpoint to the **`order-service.yml`** file in the path **`/orders`**:

```
get:
  tags:
    - order
  summary: Returns a page of orders
  operationId: findOrders
  parameters:
    - name: pageNumber
      in: query
      description: the page number, 0 is the first page
      required: true
      schema:
        type: integer
    - name: pageSize
      in: query
      description: the size of page, default is 10
      required: false
      schema:
        type: integer
        default: 10
```

```
    - name: sort
      in: query
      description: the fields on which to do the sorting
      required: false
      schema:
        type: array
        items:
          type: string
    - name: order
      in: query
      description: ascending or descending order, if the sort field is
provided
      required: false
      schema:
        type: string
        enum:
          - ASC
          - DESC
        default: ASC
    - name: status
      in: query
      description: the filter on the status
      required: false
      schema:
        type: string
  responses:
    '200':
      description: get a paginated list of orders
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/OrderPage'
    '400':
      description: Invalid input
```

```
      content:
        application/problem+json:
          schema:
            $ref: '#/components/schemas/ProblemDetail'
```

It can be seen from the request that sorting is optional and multiple fields can be provided. Also, if not specified, sorting will be done in an ascending manner.

Let us look at the response body in detail:

```
OrderPage:
  type: object
  properties:
    content:
      type: array
      items:
        $ref: '#/components/schemas/OrderPageItem'
    totalPages:
      type: integer
      description: number of total pages
    totalElements:
      type: integer
      format: int64
      description: number of total elements for current page
    first:
      type: boolean
      description: true if the current page is the first one
    last:
      type: boolean
      description: true if the current page is the last one
    number:
      type: integer
      description: number of page
OrderPageItem:
  allOf:
    - $ref: '#/components/schemas/BaseOrder'
BaseOrder:
```

```
required:
  - orderCode
  - status
type: object
discriminator:
  propertyName: type
  mapping:
    orderSummary: "#/components/schemas/OrderPageItem"
properties:
  orderCode:
    type: string
  status:
    type: string
  totalPrice:
    type: integer
    format: int64
  createdAt:
    type: string
    description: UTC date, ISO-8601 representation
    format: date-time
    example: 2024-01-01T00:00:00Z
  updatedAt:
    type: string
    description: UTC date, ISO-8601 representation
    format: date-time
    example: 2024-01-01T00:00:00Z
```

In addition to the **`content`** field, which is the list of retrieved orders, other fields on the pagination are useful to the client, such as the **`last`** field that indicates whether the current page is the last one. In addition, we used the OpenAPI inheritance feature, using the **`allOf`** field. In contrast to the previous chapter, we used the **`discriminator`** field because we planned to use, in addition to the **`OrderPageItem`** schema, another child schema for the **`findOrderByCode`** endpoint, containing more fields than the parent. To allow the client to be able to discriminate the type of serialization to be done, the discriminator field is used. If you want to know more, visit **https://swagger.io/docs/specification/data-models/inheritance-and-polymorphism/**. Generate the classes with the **`./mvnw clean compile`** command. Add the following method to the **`OrderMapper`** interface:

```
OrderPageItem toOrderPageItem(OrderEntity entity);
```

Regarding the paged query, Spring Data provides the following interface:

```
@NoRepositoryBean
public interface PagingAndSortingRepository<T, ID> extends Reposito-
ry<T, ID> {
    Iterable<T> findAll(Sort sort);
    Page<T> findAll(Pageable pageable);
}
```

If we were to do pagination without filters, we could have this interface extended to **`OrderRepository`** as well. Our paging API is more complex, however, because it involves filtering on status. Taking advantage of the naming convention, we could write the following method in **`OrderRepository`**:

```
Page<OrderEntity> findAllByStatusStartingWith(String statusStartWith,
                                              Pageable pageable);
```

Since the status parameter is evaluated with a **`startWith`**, if it is not passed by the client, we could value its server side with an empty string. This method works perfectly for our use case. However, we want to show you a more complex but more flexible alternative for other use cases. Spring Data provides the following interface:

```
public interface QueryByExampleExecutor<T> {
    <S extends T> Page<S> findAll(Example<S> example, Pageable pageable);
    //other methos
}
```

This interface supports **Queries By Example** (**QBE**). An **Example** consists of a **Probe**, which represents a domain object with only the fields useful for filtering valorized, and optionally an **`ExampleMatcher`**, which contains details on how to filter the fields of the example. Update the **`OrderRepository`** interface:

```
public interface OrderRepository extends CrudRepository<OrderEnti-
ty, String>,
        QueryByExampleExecutor<OrderEntity> {
}
```

We add to the **`OrderService`** class the **`findAll`** method, which uses the Query by Example of the **`OrderRepository`** class to retrieve a paged list of orders:

```
public OrderPage findAll(int pageNumber, int pageSize, String sta-
tus, List<String> sort, String order) {
```

```
    var example = createExample(status);
    final Page<BaseOrder> page = orderRepository.
findAll(example, pageRequest)
            .map(orderMapper::toOrderPageItem);
    return new OrderPage()
            .content(page.getContent())
            .totalElements(page.getTotalElements())
            .totalPages(page.getTotalPages())
            .first(page.isFirst())
            .last(page.isLast())
            .number(page.getNumber())
            .number(page.getNumberOfElements());
}

private Pageable createPageRequest(int pageNumber, int page-
Size, List<String> sort, String order) {
    if(sort != null) {
        Sort sortInput = Sort.by(Sort.Direction.valueOf(order),
                sort.toArray(new String[]{}));
        return PageRequest.of(pageNumber, pageSize, sortInput);
    }
    return PageRequest.of(pageNumber, pageSize);
}

private Example<OrderEntity> createExample(String status) {
    var probe = OrderEntity.builder()
            .status(status)
            .build();

    var matcher = ExampleMatcher.matching()
            .withMatcher("status", startsWith());

    return Example.of(probe, matcher);
}
```

The first statement creates an object of type **`PageRequest`**, which is an implementation of the **`Pageable`** class. The second creates an Example for **`OrderEntity`**. The probe is an object of type **`OrderEntity`**, which has only the **`status`** field value. The matcher indicates how to evaluate the **`status`** field, in this case, the static **`startWith()`** method of the **`ExampleMatcher`** interface is used. All other fields, being unvalued in the probe object, will not be used for filtering.

You can implement the **`findAll`** method of the **`OrderController`** class.

Start the application and enter more orders with the POST API created earlier. Change on the **`DB`**, the status of an order to **`DELIVERED`**, so that they are not all in the **`TAKEN_CHARGE`** status. In my case, the total number of orders is 11. Then, try making the following GET call: **`http :8081/orders pageNumber==0 pageSize==5`**. The response will be:

```
HTTP/1.1 200
{
    "content": [
        {
            "createdAt": "2024-03-31T10:55:22.581388Z",
            "orderCode": "JUEOK1711882522551",
            "status": "TAKEN_CHARGE",
            "totalPrice": 10000,
            "type": "orderSummary",
            "updatedAt": "2024-03-31T10:55:22.581388Z"
        },
        {
            "createdAt": "2024-03-31T10:55:28.000039Z",
            "orderCode": "BNXVM1711882527998",
            "status": "TAKEN_CHARGE",
            "totalPrice": 10000,
            "type": "orderSummary",
            "updatedAt": "2024-03-31T10:55:28.000039Z"
        },
        //three other elements…
    ],
    "first": true,
    "last": false,
    "number": 5,
```

```
    "totalElements": 11,
    "totalPages": 3
}
```

The response, in addition to providing the list of orders, returns the total number of items, total number of pages, and other information to the client. If we wanted to sort the orders by **`status`** and by **`createdAt`**, in a descending manner, we could make the following call:

```
http :8081/orders pageNumber==0 pageSize==11 sort==status sort==createdAt
order==DESC
```

If we also wanted to add filtering by status, for example, valued as **`DEL`**, the API should return all orders with **`DELIVERED`** status. Try making the following request:

```
http :8081/orders pageNumber==0 pageSize==11 sort==status sort==create-
dAt order==DESC status==DEL
```

If an order with **`DELIVERED`** status existed in **`DB`**, the response should look like this:

```
HTTP/1.1 200
{
    "content": [
        {
            "createdAt": "2024-03-31T19:26:42.735095Z",
            "orderCode": "C04RK1711909602708",
            "status": "DELIVERED",
            "totalPrice": 10000,
            "type": "orderSummary",
            "updatedAt": "2024-03-31T19:26:42.735095Z"
        }
    ],
    "first": true,
    "last": true,
    "number": 1,
    "totalElements": 1,
    "totalPages": 1
}
```

With Spring MVC and Spring Data, we have shown you how easy it is to create paginated APIs. Unfortunately, Spring WebFlux does not natively support pagination, since the concept of pagination conflicts with the concept of reactive. However, in **`ProductController`** class, you will find an example of pagination with Spring WebFlux.

Regarding error handling in Spring MVC, there is not much to say compared to what we have already seen in Spring WebFlux. Spring Boot version 3 provides the **`ResponseEntityExceptionHandler`** class for both WebFlux and MVC (with different packages). You can then create an ErrorHandler as you did for the previous chapter.

On logging requests and responses, you have various options, such as using filters or, if you just need to log requests and responses already serialized, you can put this property in the **`application.properties`** file: **`logging.level.org.springframework.web.servlet=TRACE`**.

In this case, the application will log both URLs and any request and response bodies, based on the **`toString`** method of the DTOs. The headers will not be logged by default. You can enable the logging of headers as well by adding the following property: **`spring.mvc.log-request-details=true`**.

# A new modern synchronous REST client: RestClient

With the release of *Spring 6* and *Spring Boot 3*, a new synchronous HTTP client called **`RestClient`**, is available in the framework. Compared to the **`RestTemplate`** client, which was the only synchronous client available by Spring until version 5, **`RestClient`** offers a more modern set of APIs that uses the Fluent approach instead of the Template pattern. We can say that **`RestClient`** is the synchronous version of **`WebClient`**, the reactive client we saw in the previous chapter shares most of the methods, as shown in *Figure 3.5*:




***Figure 3.5**: A comparison of WebClient, RestClient, and RestTemplate*

Like **`WebClient`** (and **`RestTemplate`**), **`RestClient`** needs to use an HTTP client library under the hood. Client libraries are used through the **`ClientRequestFactory`** interface. Spring already provides the following implementations for some HTTP clients:

- **`JdkClientHttpRequestFactory`** for Java's HttpClient.
- **`HttpComponentsClientHttpRequestFactory`** for use with Apache HTTP Components HttpClient.
- **`JettyClientHttpRequestFactory`** for Jetty's HttpClient.
- **`ReactorNettyClientRequestFactory`** for Reactor Netty's HttpClient.
- **`SimpleClientHttpRequestFactory`** is a simple default, which uses standard JDK facilities.

If no factory is specified, **`RestClient`** will use Apache's HttpClient or Jetty, if they are available in the classpath. Otherwise, if the **`java.net.http`** module is present in the classpath, it will use Java's HttpClient. Finally, the default factory will be used. Let us see RestClient at work by writing a **`ProductClient`** implementation that uses the Spring client to make calls to the microservice catalog service.

First, comment out the **`@Component`** annotation on the **`MockProductClient`** class to avoid having two beans of type **`ProductClient`**. In the **`config`** package, write the following class:

```
@Component
@ConfigurationProperties(prefix = "order.service")
@Data
public class OrderServiceProperties {

    private String catalogserviceUrl;
}
```

This class allows you to externalize the URL of the catalog service endpoint. In the **`application.properties`** file, add the following property:

```
order.service.catalogservice-url=http://localhost:8080/products
```

With Spring Boot, you can easily override this property with the **`ORDER_SERVICE_CATALOGSERVICE_URL`** environment variable. We create a bean of type **`RestClientCustomizer`** that allows us to generalize the RestClient configuration. In this case, we want JDK's HttpClient to always be used for each RestClient created by the bean of type **`RestClient.Builder`**:

```
@Configuration
public class RestClientConfig {

    @Bean
    RestClientCustomizer restClientCustomizer() {
        return restClientBuilder -> restClientBuilder
```

```
                .requestFactory(new JdkClientHttpRequestFactory());
    }
}
```

Also, write an exception class to handle the case when no product is found with the input product code. In the **exception** package, write the following class:

```
public class ProductOrderNotFoundException extends NestedRuntimeException {

    public ProductOrderNotFoundException(String productCode) {
        super(String.
format("Product with code %s not found", productCode));
    }
}
```

Now, we can write the implementation of **`ProductClient`**. In the **`msclient.impl`** subpackage, write the following class:

```
@Component
public class RestClientProductClient implements ProductClient {

    private final RestClient restClient;

    public RestClientProductClient(RestClient.Builder builder,
                                   OrderServiceProperties properties) {
        restClient = builder
                .baseUrl(properties.getCatalogserviceUrl())
                .build();
    }

    @Override
    public ProductResponse findByProductCode(String productCode) {
        return restClient.get()
                .uri("/" + productCode)
                .retrieve()
                .onStatus(status -> status.value() == 404,
                        (request, response) -> {
                    throw new ProductOrderNotFoundException(productCode);
```

```
                })
                .body(new ParameterizedTypeReference<>() {});
    }
}
```

Like **`WebClient`**, **`RestClient`** throws an exception if the response HTTP status is 4xx or 5xx, and, again like **`WebClient`**, we can use the **`onStatus`** method to handle these statuses. We can handle the **`ProductOrderNotFoundException`** exception in the error handler by writing the following class in the *exception* package:

```
@RestControllerAdvice
@Slf4j
public class ErrorHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(ProductOrderNotFoundException.class)
    public ResponseEntity<Object> handle(
            ProductOrderNotFoundException ex, WebRequest request) {

        log.warn(ex.getMessage());
        var pd = ProblemDetail
                .forStatusAndDetail(HttpStatus.BAD_REQUEST, ex.
getMessage());
        return new ResponseEntity<>(pd, HttpStatus.BAD_REQUEST);
    }
}
```

Start the order service and catalog service applications. From pgadmin, we recommend you delete all rows in the **`orders`** and **`order_items`** tables to have a clean situation. Create two products by invoking the catalog service POST API:

```
http :8080/products code=0001 name="ALaptop" category=lap-
top price=60000 brand=FirstBrand
http :8080/products code=0002 name="SuperLaptop" category=lap-
top price=30000 brand=FirstBrand
```

Try creating an order with these two products:

```
http :8081/orders \
  Content-Type:application/json \
  "products[0][productCode]"=0001 \
```

```
  "products[0][quantity]":=2 \
  "products[1][productCode]"=0002 \
  "products[1][quantity]":=1
HTTP/1.1 201
Location: http://localhost:8081/orders/AL7ZS1712395957445
```

Invoke the **`findOrderByCode`** API: **`http :8081/orders/AL7ZS1712395957445`**. You should get a response like this:

```
HTTP/1.1 200
{
    "createdAt": "2024-04-06T09:32:37.478903Z",
    “orderCode”: “AL7ZS1712395957445”,
    “products”: [
        {
            “productCode”: “0001”,
            “quantity”: 2
        },
        {
            "productCode": "0002",
            "quantity": 1
        }
    ],
    "status": "TAKEN_CHARGE",
    "totalPrice": 150000,
    "type": "orderDetail",
    "updatedAt": "2024-04-06T09:32:37.478903Z"
}
```

The total price is consistent with the products purchased since the order contains two products with code 0001 (where the price is 60000 per single piece) and one product with code 0002 (where the price is 30000). Try creating an order with an existing product and a non-existing product:

```
http :8081/orders \
  Content-Type:application/json \
  "products[0][productCode]"=0001 \
  "products[0][quantity]":=2 \
```

```
  "products[1][productCode]"=0003 \
  "products[1][quantity]":=1
```

You will get the following response:

```
HTTP/1.1 400
{
    "detail": "Product with code 0003 not found",
    "instance": "/orders",
    "status": 400,
    "title": "Bad Request",
    "type": "about:blank"
}
```

If you want to create an order, it must contain all existing products. Our **`addOrder`** API is now consistent with the product master.

In this section, we have shown you how RestClient works directly, however, to manage HTTP clients as a catalog service you also have other ways available, such as autogenerating client APIs with the OpenAPI Generator plugin or using RestClient transparently by using the HTTP interfaces provided by Spring (take a look at **https://docs.spring.io/spring-framework/reference/integration/rest-clients.html#rest-http-interface**).

# Testing in Spring MVC

Spring MVC provides a testing framework called **MockMvc** for testing controllers using a mock server instead of a real running server. MockMvc can be used on its own to execute requests and test the responses returned by the controller, or it can be used with WebTestClient (which we have already seen in the previous chapter), which brings the advantage of working with more high-level objects and performs more comprehensive HTTP tests. We will show you the use of MockMvc with WebTestClient. In the test folder, create the api package and write the following test class:

```
@WebMvcTest(OrderController.class)
class OrderControllerTest {

    @MockBean
    private OrderService orderService;

    @Autowired
    private WebTestClient webTestClient;
```

```
    @Test
    void addOrderOkTest() {
        var request = new OrderRequest()
                .products(List.of(
                        new ProductOrder("code1", 1)
                ));
        when(orderService.addOrder(request)).thenReturn("code1");
        webTestClient.post()
                .uri("/orders")
                .bodyValue(request)
                .exchange()
                .expectStatus().isCreated()
                .expectBody(Void.class)
                .consumeWith(result ->
                        assertThat(result.getResponseHeaders()
                                .get("Location").get(0))
                        .isEqualTo("http://localhost/orders/code1"));

    }
}
```

As you can see, SpringMVC controllers test in the same way as Spring WebFlux controllers. It changes the annotation that allows WebTestClient injections, which in Spring MVC is **`@WebMvcTest`** while in WebFlux it is **`@WebFluxTest`**.

In order to test the ProductClient HTTP client, you can use the **`MockRestServiceServer`** test class that is provided by Spring. It gives a way to set up expected requests that will be performed through the RestTemplate or RestClient as well as mock responses to send back thus removing the need for an actual server. In the test folder, create the **`middleware.msclient`** package and write the following class:

```
@RestClientTest(ProductClient.class)
@Import(OrderServiceProperties.class)
class ProductClientTest {

    @Autowired
    private MockRestServiceServer server;
```

```
    @Autowired
    private ProductClient productClient;

    @Autowired
    private ObjectMapper objectMapper;

    @Test
    void findByProductCodeOk() throws JsonProcessingException {
        var expectedResponse = new ProductResponse("code1", "laptop", 10000L);
        var responseJson = objectMapper.writeValueAsString(expectedResponse);
        server.expect(requestTo("http://localhost:8080/products/code1"))
                .andRespond(
                        withSuccess(responseJson, MediaType.APPLICATION_JSON)
                );

        var actualResponse = productClient.findByProductCode("code1");
        assertThat(actualResponse).isEqualTo(expectedResponse);
    }
}
```

In order to inject the MockRestServiceServer bean, the test class must be annotated with **`@RestClientTest`**.

# Spring MVC performance comparison: Platform vs. virtual threads

Optimizing concurrency in Spring MVC applications is crucial in modern software development, especially as applications handle increasingly complex workloads. Traditionally, Spring MVC uses platform threads to implement the thread-per-request paradigm. However, we have seen that with Java 21 and Spring Boot 3 it is easy to switch to virtual threads, which promises enhanced efficiency. To assess the practical benefits of virtual threads over platform threads, we conducted a detailed benchmark test using Apache Benchmark (**https://httpd.apache.org/docs/2.4/programs/ab.html**), focusing on key performance indicators such as total test duration, requests per second, average request time, and data transfer rate.

The application on which the tests were run is a Spring Boot app using Spring MVC. The application is very basic, it provides an endpoint that returns ten records of type book. There are no interactions with databases or external tools. The code is as follows:

```
@SpringBootApplication
public class VtApplication {

	public static void main(String[] args) {
		SpringApplication.run(VtApplication.class, args);
	}

	@RestController
	public class MockApi {

		@GetMapping
		public ResponseEntity<List<Book>> findAll()
					throws InterruptedException {
			var books = new ArrayList<Book>();
			Thread.sleep(500L);
			for(int i = 0; i < 10; i++) {
				books.add(new Book(i, UUID.randomUUID().toString()));
			}
			return ResponseEntity.ok(books);
		}
	}

	record Book(int bookId, String bookName){}

}
```

The same application was tested using both platform threads and virtual threads by changing the value of the **`spring.threads.virtual.enabled`** property. The load test involved 10000 total requests with a competition factor of 250. The application was released as a JAR package in an Amazon EC2 machine t3.micro (2 vCPUs and 1 GiB of memory).

The test results are summarized in the *Table 3.2*:

| Metric | Platform Threads | Virtual Threads | Better Value |
|---|---|---|---|
| **Time taken for tests (s)** | 26.538 | 22.855 | Lower is better |
| **Requests per second (#/sec)** | 376.82 | 437.54 | Higher is better |
| **Time per request (ms, mean)** | 663.44 | 571.378 | Lower is better |
| **Transfer rate (Kbytes/sec)** | 270.84 | 314.48 | Higher is better |

***Table 3.2:***

The results are compelling. Virtual threads completed the test suite in 22.855 seconds, outperforming platform threads, which required 26.538 seconds, for a similar load. This reduction in total test duration is complemented by an increase in throughput, with virtual threads achieving 437.54 requests per second, compared to 376.82 with platform threads. Additionally, the mean time per request was significantly lower for virtual threads (571.378 ms versus 663.440 ms), indicating faster response handling under concurrent conditions. Lastly, data transfer rates were markedly improved, with virtual threads achieving a transfer rate of 314.48 Kbytes/sec against 270.84 Kbytes/sec for platform threads.

These findings suggest that virtual threads offer substantial advantages for high-concurrency Spring MVC applications, enhancing both processing speed and data handling efficiency. By adopting virtual threads, developers can achieve better scalability and responsiveness, making this approach an invaluable option for applications that demand high performance and seamless concurrency management.

# Choose between virtual threads and WebFlux

After reading this chapter and the previous one, you may be wondering which approach to use in your applications, the thread-per-request approach with Spring MVC and virtual threads or the event loop approach with Spring WebFlux. If your application was already written with Spring MVC, and you are looking for ways to optimize thread management, using virtual threads allows you not to heavily refactor your code (just upgrade to Spring Boot 3, Java 21, and add the property to enable virtual threads). If you are starting from scratch, the choice depends on your team's knowledge. With Spring WebFlux, you can write code functionally, and handle backpressure, and you are not forced to migrate to Java 21. However, the reactive approach is much more complex than the traditional one, and although from some benchmarks Spring WebFlux proves to be slightly more performant at this time, virtual threads are to be preferred because of the simplicity of design associated with scalability performance. If your application already uses WebFlux, we do not recommend moving to Spring MVC. If you are interested in more details on comparing the performance of Spring WebFlux and Spring MVC with virtual threads, you can check out this article: **https://www.vincenzoracca.com/en/blog/framework/spring/virtual-threads-vs-webflux**.

# Conclusion

In this chapter, we saw what virtual threads are in detail and explained why they are an important feature in Java. Through virtual threads, you can use the *simple* thread-per-request model with the same scalability as asynchronous programming. Then we showed you how to create REST APIs traditionally with Spring MVC, and we explored how Spring makes it easy to switch usage to virtual threads with a simple property. You have also seen the new synchronous HTTP client, RestClient, which allows you to use the same fluent APIs as WebClient. With this chapter, you have the means to write scalable REST APIs by

choosing between Spring MVC and Spring WebFlux. In the next chapter, we will look at a different approach to writing APIs using GraphQL.

# Points to remember

- Spring MVC used the thread-per-request approach and the Front Controller pattern.
- Spring MVC by default does not use virtual threads. These must be enabled by property.
- Use @WebMvcTest to test REST APIs with Spring MVC.

# Exercises

1. Try implementing GET, PUT, and DELETE endpoints for the order resource yourself. Allow modification and deletion of an order only if its status is taken charge.
2. Try implementing tests for these other endpoints.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

**https://discord.bpbonline.com**



# Chapter 4
# GraphQL with Spring Boot

## Introduction

In this chapter, we delve into the innovative world of GraphQL, a powerful query language for APIs, and its integration with the Spring framework, *Spring for GraphQL*. This chapter sets the stage for a deeper understanding of the essentials of GraphQL, from its core principles to advanced features, ensuring a solid foundation for implementing highly efficient APIs. Starting with the fundamentals, we explore what makes GraphQL a unique and compelling choice for modern API design, contrasting it with traditional REST APIs to highlight its advantages in flexibility and efficiency. This chapter covers the basic structure of GraphQL queries, mutations, and the integral role of schemas in defining how data can be accessed and manipulated. Moving forward, the focus shifts to the practical implementation of GraphQL APIs using Spring for GraphQL. Each step is accompanied by code snippets and configuration examples to guide you through the process of setting up a GraphQL server. To ensure the reliability of the GraphQL services you build, we also cover testing strategies tailored specifically for GraphQL APIs. Finally, the chapter rounds off with a comprehensive comparison between GraphQL and REST, providing you with a clear understanding of when and why to use each approach. By the end of this chapter, you will be well-prepared to harness the power of GraphQL in conjunction with Spring Boot to create cutting-edge, efficient, and scalable web applications.

# Structure

In this chapter, we will discuss the following topics:

- Fundamentals of GraphQL
- Implementing GraphQL APIs with Spring Boot
- Testing GraphQL APIs
- Comparing REST and GraphQL

# Objectives

The primary objective of this chapter is to equip readers with a thorough understanding of GraphQL and its integration with Spring for GraphQL. We aim to demystify the foundational elements of GraphQL, including its operational mechanics, such as queries, mutations, and schema definitions. By illustrating these concepts through practical examples, we intend for readers to grasp not only the theoretical aspects but also the practical implementation of GraphQL within a Spring Boot application. Furthermore, the chapter seeks to enhance the reader's competency in developing, testing, and optimizing GraphQL APIs. Lastly, by comparing GraphQL with traditional REST APIs, we strive to provide a clear perspective on choosing the appropriate technology based on specific project requirements and scenarios, thereby empowering readers to make informed decisions in their future API development endeavors.

# Fundamentals of GraphQL

GraphQL is a query language for API and a server-side runtime for performing such queries on data. Unlike the REST architecture, GraphQL allows clients to request exactly the data they need, thus reducing the number of requests and the amount of data transferred. It is an alternative to REST, SOAP, or gRPC.

Like REST, GraphQL is independent of both the programming language and the transport protocol used, although the most popular one is HTTP.

The GraphQL server is the one that processes and returns the results of queries made by clients. The result can be obtained by aggregating data from multiple sources, completely transparent to the client. For example, in *Figure 4.1*, the GraphQL server returns the response by aggregating data from a REST service, a database, and a SOAP service:

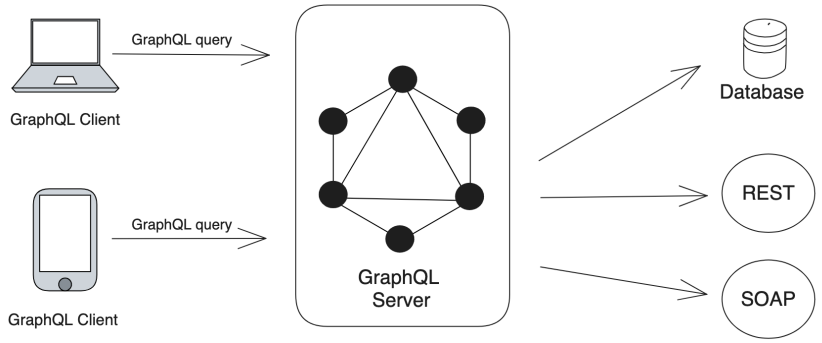



***Figure 4.1**: An example of a GraphQL server retrieving data from different sources*

# Schemas and types

A GraphQL service is created by first defining one or more schema files, which contain the definition of the data types handled by the server. A GraphQL schema definition that handles books might be similar to the one shown below:

```
type Book {
    id: ID
    name: String
    pageCount: Int
    author: Author
}

type Author {
    id: ID
    firstName: String
    lastName: String
}
```

You can see from the example that field types can refer to scalars, such as the name field, which is of type String, or to other types, such as the **`author`** field which is of type Author. The scalar types available by default in GraphQL are listed in *Table 4.1*:

| | |
|---|---|
| Int | A signed 32□bit integer |
| Float | A signed double-precision floating-point value |
| String | A UTF□8 character sequence |
| Boolean | Represents the following values: true, false |
| ID | A unique identifier, serialized in the same way as a String |

***Table 4.1:***

In most implementations of GraphQL services, you can also specify custom scalar types, such as to map dates or 64-bit integer values (Long).

Fields can be *nullable* or *non-nullable*. To indicate a *non-nullable* field, just put the suffix "!" on the type of the field, for example, if we wanted to make the name field of Book mandatory, the field definition would become **`name: String!`**.

In order to declare a field of type list, just wrap the type in square brackets. If we assume that a book might have multiple authors, the **`author`** field of the Book type would be declared in this way:

```
author: [Author]
```

Each field can have zero or more arguments. An argument can be mandatory or optional. An optional argument can have a default value. Let us look at the following example:

```
type Starship {
    id: ID!
    name: String!
    length(unit: String = "METER", anotherArg: String!): Float
}
```

The **`length`** field of the **`Starship`** type has two arguments, the first optional, unit, which defaults to **`METER`**, while the second argument, **`anotherArg`**, is mandatory.

Arguments can also be complex objects and not just scalar values. In GraphQL, there is the input type, which is defined by the keyword **`input`** instead of **`type`**, as shown below:

```
input BookInput {
    name: String,
    pageCount: Int
    authorId: String
}
```

The input type is often used as the argument of mutations.

# Queries and mutations

There are three special types **query**, **mutation**, and **subscription**. Each GraphQL service includes the query type and may also include mutation and subscription. These types have the same characteristics as the *normal* types seen in the previous section, but they are called special because they define the entry point for requests on the GraphQL server. In GraphQL, the client defines a request by an operation, which may include one or more queries. An example of an operation containing two queries is as follows:

```
{
  bookById(id: "book-1") {
    name
  }
  booksByName(name: "Spring Boot 3 API Mastery") {
    id,
    author {
      firstName
      lastName
    }
  }
}
```

Specifically, the parameters in round brackets are the arguments passed as input, while the fields in curly brackets are the fields that the client is interested in receiving as output. We could define the parameters in curly brackets as the projections of SQL queries. Due to this feature, the client requests only the output fields it needs from the server, avoiding over-fetching. It is possible to perform the above query only if we define in the schema, the **`Query`** type having the following two fields, **`booksById`** and **`booksByName`**:

```
type Query {
    bookById(id: ID): Book
    booksByName(name: String): [Book]
}
```

The result of the previous operation could be similar to the following code:

```
{
  "data": {
    "bookById": {
      "name": "Spring Boot 3 API Mastery"
    },
    "booksByName": [
      {
        "id": "book-1",
        "author": {
          "firstName": "Vincenzo",
          "lastName": "Racca"
```

```
        }
      }
    ]
  }
}
```

It is possible and advisable to define an operation name on the query that is being executed, to eliminate any ambiguity. For example, we could redefine the previous query as follows:

```
query bookOperations {
  bookById(id: "book-1") {
    name
  }
  booksByName(name: "Spring Boot 3 API Mastery") {
    id,
    author {
      firstName
      lastName
    }
  }
}
```

**Note: The `bookOperations` operation executes two queries, `bookById`, and `booksByName`; we could also have executed the two queries with two separate requests, but one of the advantages of GraphQL is that we can execute multiple queries (in this case, `bookById` and `booksByName`) by making a single request (named, in this case, bookOperations). This brings the advantage of not needing to make multiple calls to the API to perform multiple queries.**

The purpose of queries is to fetch data. To perform insert, modify, and delete operations, mutations are used. Nothing detracts from the fact that you can also modify data with queries, but it is not good practice, just like the GET API in REST, which should be used only for fetching data and not for modifying it.

The syntax of mutations is like that of queries. We define in the schema a mutation to save a new book, as shown below:

```
type Mutation {

    saveBook(name: String, pageCount: Int, authorId: String): Book

}
```

The request that performs the mutation could be as follows:

```
mutation CreateBook {
  saveBook(name: "NewBook", pageCount: 100, authorId: "author-1") {
    id
  }
}
```

The response will output only the ID of the newly created book. The response will then be as follows:

```
{
  "data": {
    "saveBookOld": {
      "id": "book-4"
    }
  }
}
```

Instead of creating a mutation that takes as input all these scalars (**`name`**, **`pageCount`**, and **`authorId`**), we could create a mutation having as arguments an input type that wraps them. The **`saveBook`** mutation could become:

```
type Mutation {
    saveBook(book: BookInput!): Book
}

input BookInput {
    name: String,
    pageCount: Int
    authorId: String
}
```

When using the input type in the request, it would look like this:

```
mutation CreateBook($bookInput: BookInput!) {
  saveBook(book: $bookInput) {
    id
  }
}
```

In the example shown above, we used a variable called **bookInput**. Variables allow you to pass arguments with dynamic values. The definition of the variable **bookInput** is as follows:

```
{
  "bookInput": {
    "name": "NewBook",
    "pageCount": 100,
    "authorId": "author-1"
  }
}
```

An important difference between queries and mutations is that if a request contains multiple queries, they can be processed in parallel by the server. Conversely, if a request contains multiple mutations, they will be processed one at a time.

Another special type in GraphQL is **Subscription**, which allows the client to subscribe to a data stream. As with mutations, the definition of subscriptions is like that of queries, as shown below:

```
type Subscription {
    subscribe: Book
}
```

Subscriptions are especially useful when using protocols other than HTTP, such as **WebSocket** and **RSocket**. If you want to learn more about the theoretical aspects of GraphQL, we recommend you to look at the official document: **https://graphql.org/learn**.

# Implementing GraphQL APIs with Spring Boot

The Spring for GraphQL module, successor to the GraphQL Java Spring module provides GraphQL support for Spring applications. In the previous chapter, we designed the order service microservice containing the APIs, of **findOrderByCode**. The latter takes an order code as input and returns the order detail with the list of product codes and quantities. The response payload has the following format:

```
{
    "createdAt": "2024-04-07T14:38:48.058763Z",
    “orderCode”: “6M6ZN1712500728019”,
    “products”: [
        {
```

```
            “productCode”: “0001”,
            “quantity”: 2
        },
        {
            "productCode": "0002",
            "quantity": 1
        }
    ],
    "status": "DELIVERED",
    "totalPrice": 150000,
    "type": "orderDetail",
    "updatedAt": "2024-04-07T14:38:48.058763Z"
}
```

If the client wanted the details of each product, such as name and brand, they would have to call for each product code in the previous payload, the **`findProductByCode`** API of the catalog service. We give this responsibility to the new GraphQL service microservice, which leveraging GraphQL features, retrieves all the information requested by the client, including product details if requested by aggregating the responses from the **`findOrderByCode`** and **`findProductByCode`** APIs. We can summarize the operation of the new microservice by noticing *Figure 4.2*:

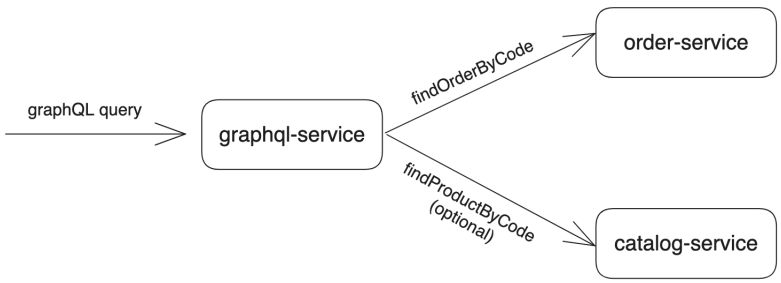



***Figure 4.2**: The new graphql service microservice that acts as a graphql-server*

The code for this chapter is available in the **`chapter-04`** folder of the project's GitHub repository. Also, in the package book, you will find the Java code inherent to the examples

given in the previous paragraph. Let us create the project skeleton from Spring Initializr as shown in *Figure 4.3*:

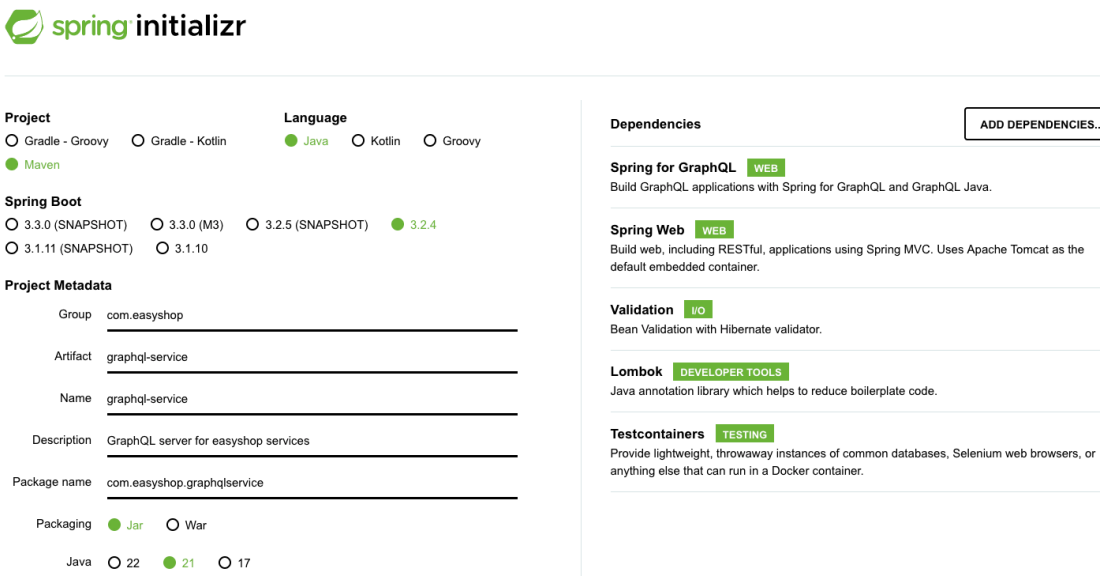



***Figure 4.3**: Spring Initializr for GraphQL.*

Spring for GraphQl is independent of the transport protocol used, so to work, we need to provide the dependency of Spring MVC, Spring WebFlux, Spring WebSocket, or Spring RSocket. For this project, we choose Spring MVC and will take advantage of virtual threads.

By downloading the project, you will find the empty **`graphql`** folder inside the resources folder. Spring by default reads GraphQL schemas (files with the graphqls extension) from that folder. To use custom scalars without writing additional code, we import the **`graphql-java-extended-scalars`** library into the pom.xml file, as shown below:

```
<dependency>
    <groupId>com.graphql-java</groupId>
    <artifactId>graphql-java-extended-scalars</artifactId>
    <version>21.0</version>
</dependency>
```

In GraphQL, you can create a hierarchical schema structure using the keyword extend. We write the parent file, **`schema.graphqls`**, in the **`graphql`** folder, as shown below:

```
scalar DateTime
scalar Long
type Query {
}
```

```
type Mutation {
}
```

The keyword scalar allows us to declare custom scalars. In the same folder, we write the file **`order.graphqls`**, for declaring Order, Product types, and the queries associated with them, as shown below:

```
extend type Query {
    findOrderByCode(orderCode: ID): Order
}

type Order {
    orderCode: ID
    status: String
    totalPrice: Long
    createdAt: DateTime
    updatedAt: DateTime
    products: [Product]
}

type Product {
    productCode: ID
    name: String
    category: String
    price: Int
    brand: String
    quantity: Int
}
```

We write the **`Product`** and **`Order`** classes that map to GraphQL schemas. Create the **`dto`** package and write the following Java records:

```
@Builder(toBuilder = true)
public record Order(String orderCode, String status, Long totalPrice,
                    OffsetDateTime createdAt, OffsetDateTime updatedAt,
                    List<Product> products) {}
```

```
@Builder(toBuilder = true)
public record Product(String productCode, String name, String category,
                      Long price, String brand, Integer quantity) {}
```

Custom scalars must be registered using the Spring **`RuntimeWiringConfigurer`** class. We create the **`config`** package and write the following class to register the custom scalars:

```
@Configuration
public class GraphQlConfig {

    @Bean
    RuntimeWiringConfigurer runtimeWiringConfigurer() {
        return wiringBuilder -> wiringBuilder
                .scalar(ExtendedScalars.DateTime)
                .scalar(ExtendedScalars.GraphQLLong);
    }
}
```

In the same package, we will create the class that allows us to define the catalog service and order service endpoints, just as we did in the previous chapter:

```
@Component
@ConfigurationProperties(prefix = "graphql.service")
@Data
public class GraphQlServiceProperties {

    private String catalogserviceUrl;
    private String orderserviceUrl;
}
```

Now, we will use the **`RestClient`** class to make HTTP calls to the catalog service and order service. However, this time we are using HTTP interfaces so that we can demonstrate the approach we only mentioned in the previous chapter. Let us create the **`middleware.msclient`** package and write the interface that will make the call to order service:

```
public interface OrderClient {

    @GetExchange("/{orderCode}")
    Order findOrderByCode(@PathVariable String orderCode);
}
```

The **`@GetExchange`** annotation tells Spring that it will have to make an HTTP call with a GET method. HTTP interface method parameters can be annotated with the same annotations used in **`RestController`** methods, such as **`@PathVariable`**. The interface will be implemented by a proxy instance created with Spring's **`HttpServiceProxyFactory`** class. Let us create a configuration class inside the **`config`** package to handle the process mentioned above:

```
@Configuration
public class RestClientsConfig {

    @Bean
    OrderClient orderClientNew(RestClient.Builder builder,
                               GraphQlServiceProperties properties) {
        RestClient restClient = builder
                .baseUrl(properties.getOrderserviceUrl())
                .build();

        var adapter = RestClientAdapter.create(restClient);
        var factory = HttpServiceProxyFactory.builderFor(adapter).build();
        return factory.createClient(OrderClient.class);
    }
}
```

The **`ProductClient`** interface and the configuration of its factory instance are the same as **`OrderClient`**, so we leave the implementation to you.

Create the service package and write the **`OrderService`** class that calls the two HTTP interfaces, as shown below:

```
@Service
@RequiredArgsConstructor
@Slf4j
public class OrderService {

    private final OrderClient orderClient;
    private final ProductClient productClient;

    public Order findOrderByCode(String orderCode) {
        log.info("HTTP fetching order with orderCode: {}", orderCode);
        return orderClient.findOrderByCode(orderCode);
```

```
    }

    public List<Product> getProductsFromOrder(Order order) {
        return order.products().stream()
                .map(this::getProductDetail)
                .toList();
    }

    private Product getProductDetail(Product productSummary) {
        log.info("HTTP fetching product with productCode: {}",
                productSummary.productCode());
        var product = productClient.findProductByCode(productSummary
                .productCode());
        return product.toBuilder()
                .productCode(productSummary.productCode())
                .quantity(productSummary.quantity())
                .build();
    }
}
```

The **findOrderById** method is trivial, calling the client method without adding logic. The **getProductsFromOrder** method takes an order as input, which contains the list of product summaries (product code and quantity), and for each product code, it calls the catalog service API to retrieve the details, such as **name** and **brand**. You may wonder why this method is not called in the **findOrderByCode** method but is separate. We will show you by writing the controller class.

Similar to Spring MVC and Spring WebFlux, Spring for GraphQL allows you to use the **@Controller** annotation to declare request handlers. Specifically, Spring for GraphQL uses the **RuntimeWiring.Builder** class to register each controller method as a GraphQL **graphql.schema.DataFetcher**. A **DataFetcher** provides the logic to retrieve data from a query or any field in the schema. Create the **api** package and write the **OrderController** class, as shown below:

```
@Controller
@RequiredArgsConstructor
@Slf4j
public class OrderController {

    private final OrderService orderService;
```

```
    @QueryMapping
    public Order findOrderByCode(@Argument String orderCode) {
        log.info("Call findOrderByCode with orderCode: {}", orderCode);
        return orderService.findOrderByCode(orderCode);
    }

    @SchemaMapping
    public List<Product> products(Order order) {
        log.info("Call products from orderCode: {}", order.orderCode());
        return orderService.getProductsFromOrder(order);
    }
}
```

The **`@Argument`** annotation allows GraphQL arguments to be binded to Java objects. If the binding fails, the **`BindException`** is thrown.

The **`@QueryMapping("queryName")`** annotation allows a GraphQL query to be associated with the handler method. If the query name is not specified in the annotation, then Spring will associate the method with the query having the same method name. Also, **`@QuaryMapping`** is a wrapper of **`@SchemaMapping(typeName = "Query")`**. The same is true for methods that map mutations and subscriptions. In fact, the **`@MutationMapping`** and **`@SubscriptionMapping`** annotations wrap **`@SchemaMapping(typeName="Mutation")`** and **`@SchemaMapping(typeName="Subscription")`**, respectively.

The **`@SchemaMapping(typeName="", field="")`** annotation maps a handler's method to a GraphQL schema field. If the **`typeName`** and field arguments are not specified in the annotation, then the **`typeName`** will automatically be associated with the class name of the method argument, while the field value will be associated with the method name.

Spring provides the GraphiQL graph client to easily test the GraphQL API, which by default is not enabled. To enable it, we need to set the property **`spring.graphql.graphiql.enabled=true`** in the **`application.properties`** file. The file will have the following content:

```
spring.application.name=graphql-service
spring.graphql.graphiql.enabled=true
spring.graphql.schema.printer.enabled=true
server.port=8082
spring.threads.virtual.enabled=true
graphql.service.catalogservice-url=http://localhost:8080/products
graphql.service.orderservice-url=http://localhost:8081/orders
```

Start the **`graphql`** service, catalog service, and order service microservices (do not forget to start the Postgres container as well). From a browser, type the URL **`http://localhost:8082/graphiql`**, and you will see the GraphiQL UI as shown in *Figure 4.4*:

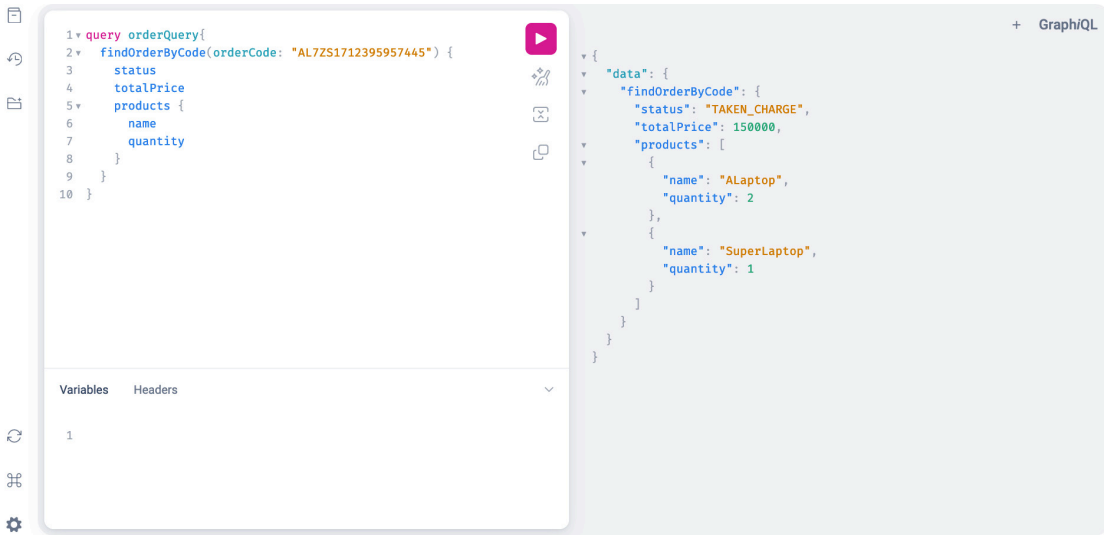



***Figure 4.4**: The UI of GraphiQL*

The GraphiQL client will call the server endpoint, which by default is **`/graphql`**. If the product and order DB tables are empty, insert records, then perform the following query on an existing order code:

```
query orderQuery{
  findOrderByCode(orderCode: "AL7ZS1712395957445") {
    status
    totalPrice
    products {
      name
      quantity
    }
  }
}
```

You will get a result similar to the following code:

```
{
  "data": {
    "findOrderByCode": {
```

```
      "status": "TAKEN_CHARGE",
      "totalPrice": 150000,
      "products": [
        {
          "name": "ALaptop",
          "quantity": 2
        },
        {
          "name": "SuperLaptop",
          "quantity": 1
        }
      ]
    }
  }
}
```

Let us take a look at the logs:

```
Call findOrderByCode with orderCode: AL7ZS1712395957445
HTTP fetching order with orderCode: AL7ZS1712395957445
Call products from orderCode: AL7ZS1712395957445
HTTP fetching product with productCode: 0001
HTTP fetching product with productCode: 0002
```

As we expected, first the **`findOrderByCode`** method is invoked which retrieves the order detail from the order service HTTP call, and then the handler's products method is invoked which retrieves the detail of the two products by calling the catalog service. Let us try the same query, but without the **`products`** field:

```
query orderQuery{
  findOrderByCode(orderCode: "AL7ZS1712395957445") {
    status
    totalPrice
  }
}
```

The response will be similar to the previous one, without the **`products`** field. Let us look at the logs:

```
Call findOrderByCode with orderCode: AL7ZS1712395957445
```

Since the **products** field is not required in the query, the **products** method is not called. This is the reason why the previous service methods are distinct, they are associated with two different **DataFetchers**. If the products field is not required by the query, the **DataFetcher** of the products field will not be called.

# GraphQL N + 1 problem

The N+1 problem in GraphQL is a performance issue that occurs when a query results in multiple, unnecessary database or API requests. This happens when a query requests a list of items that have related data, and each individual item's related data is fetched in a separate request. Let us look at an example of this problem right away.

We add another query that allows us to retrieve the list of orders, filtered possibly by following status (filter of type "startWith"):

```
extend type Query {
    findOrderByCode(orderCode: ID): Order
    orders(status: String): [Order]
}
```

In order to retrieve the list of orders, **graphql** service will have to invoke the **findOrders** API of the order service. For each order retrieved, it will have to perform the same logic as the **findOrderByCode** query. Create the **dto** subpackage in **middleware.msclient** and write the DTO class that maps the following **findOrders** API response:

```
@Builder(toBuilder = true)
public record OrderPage(List<Order> content) {
}
```

For simplicity, we map only the response's order list, thus excluding pagination management. In **OrderClient**, add the method that calls the API, as shown below:

```
@GetExchange
OrderPage findOrders(@RequestParam Integer pageNumber,
                     @RequestParam Integer pageSize,
                     @RequestParam(required = false) String status);
```

We also add the **findOrders** method in **OrderService** which simply invokes the method shown above and for each order retrieved, calls the **findOrderByCode** method, refer to the following code for clarity:

```
public Collection<Order> findOrders(String status) {
    var orderPage = orderClient.findOrders(0, 50, status);
    return orderPage.content().parallelStream()
```

```
            .map(order -> findOrderByCode(order.orderCode()))
            .toList();

}
```

Finally, we add the controller method that handles the new query:

```
@QueryMapping
public Collection<Order> orders(@Argument String status) {
    log.info("Call orders with status: {}", status);
    return orderService.findOrders(status);
}
```

Now, all we have to do is run the queries. Let us try the one shown below:

```
query orderQuery{
  orders(status: "TAK") {
    orderCode
    products {
      productCode
      name
    }
  }
}
```

The response will be like this:

```
{
  "data": {
    "orders": [
      {
        "orderCode": "AL7ZS1712395957445",
        "products": [
          {
            "productCode": "0001",
            "name": "ALaptop"
          },
          {
            "productCode": "0002",
```

```
            "name": "SuperLaptop"
          }
        ]
      },
      {
        "orderCode": "G2JVQ1712426415616",
        "products": [
          {
            "productCode": "0001",
            "name": "ALaptop"
          }
        ]
      },
      {
        "orderCode": "JM6JT1712426904072",
        "products": [
          {
            "productCode": "0001",
            "name": "ALaptop"
          }
        ]
      }
    ]
  }
}
```

Note that all three orders contain the product with code 0001. Let us analyze the logs, leaving out those related to the HTTP calls to retrieve the order details:

```
Call orders with status: TAK
Call products from orderCode: AL7ZS1712395957445
HTTP fetching product with productCode: 0001
HTTP fetching product with productCode: 0002
Call products from orderCode: G2JVQ1712426415616
HTTP fetching product with productCode: 0001
Call products from orderCode: JM6JT1712426904072
HTTP fetching product with productCode: 0001
```

We can see that for each order, an HTTP call is made to retrieve the product details. Also, although some orders share the same product, HTTP calls are made multiple times to retrieve the same product, the one with code 0001. This problem is called the **N+1 problem**.

GraphQL's batch loading allows this problem to be solved through the use of DataLoader, which allows values to be grouped and cached for each request. Spring for GraphQL provides the **`@BatchMapping`** annotation to implement batch loading. Annotated methods should return Mono<Map<K, V>>, Map<K, V>, Flux<V>, or Collection<V>.

In the **`chapter-04/catalog-service`** folder, we have modified the **`findProducts`** API so that we can retrieve products by filtering by product code. Now, we will create the classes that map the response of this API to the **`middleware.msclient.dto`** package:

```
public record ProductPage(List<ProductResponse> content) {
}


public record ProductResponse(String code, String name, String category,
                              Long price, String brand) {
}
```

Then, add the **`findProducts`** method in the **`ProductClient`** class, which will allow us to invoke the catalog service API, as shown below:

```
@GetExchange
ProductPage findProducts(@RequestParam Integer page, @
RequestParam Integer size,
                         @RequestParam Collection<String> productCode);
```

Let us create a new method in the **`OrderService`** class that gives a list of orders as input, retrieves all products with a single HTTP call, and returns a map with the key to the order and value of its list of products, as intended by the **`@BatchMapping`** specification:

```
public Map<Order, List<Product>> getProductsFromOrders(List<Order> orders) {
    var productCodes = orders.stream()
            .flatMap(order -> order.products().parallelStream()
                    .map(Product::productCode))
            .collect(Collectors.toSet());

    log.info("HTTP fetching products with productCodes: {}", productCodes);
    var products = productClient.findProducts(0, productCodes.size(),
            productCodes).content();

    var mapProduct = products.stream()
```

```
            .collect(Collectors.toMap(ProductResponse::code, product -> product));

    return orders.stream()
            .collect(Collectors.toMap(
                    order -> order,
                    order -> order.products()
                            .parallelStream()
                            .map(product ->
                                    buildProduct(product,
                                            mapProduct.get(product.productCode()))
                            )
                            .toList()

            ));
}

private Product buildProduct(Product product, ProductResponse produc-
tResponse) {
    //enrich product with productResponse
}
```

Finally, we will modify the **`product`** method of the **`OrderController`** class to take advantage of batch loading:

```
@BatchMapping
public Map<Order, List<Product>> products(@Argument List<Order> orders) {
    log.info("Call products from orderCodes: {}", orders.stream().
map(Order::orderCode));
    return orderService.getProductsFromOrders(orders);
}
```

If you re-run the previous query, you will have the same result, but the processing will be more efficient because the four HTTP calls to retrieve individual products are replaced by a single call, taking advantage of batch loading that groups all the orders in the result of the query orders.

# Error handling in Spring for GraphQL

Centralized exception handling in Spring for GraphQL is similar to what we have already seen for Spring MVC and Spring WebFlux. You can annotate a class with **`@ControllerAdvice`**,

which contains methods annotated with **`@GraphQlExceptionHandler`** to handle any type of exception. Create the exception package and copy the **`OrderNotFoundException`** class seen in the previous chapter. Use it in the **`findOrderByCode`** method of **`OrderService`**, as shown below:

```
public Order findOrderByCode(String orderCode) {
    try {
        log.info("HTTP fetching order with orderCode: {}", orderCode);
        return orderClient.findOrderByCode(orderCode);
    }
    catch (HttpClientErrorException.NotFound nfe) {
        throw new OrderNotFoundException(orderCode);
    }
}
```

After that, in the exception package, write the **`ErrorHandler`** class responsible for centralized exception handling, as shown below:

```
@ControllerAdvice
public class ErrorHandler {

    @GraphQlExceptionHandler
    public GraphQLError handle(OrderNotFoundException ex) {
        return GraphQLError.newError()
                .errorType(ErrorType.NOT_FOUND)
                .message(ex.getMessage())
                .build();
    }
}
```

The **`GraphQLError`** class implements the specification described in the GraphQL documentation: **https://spec.graphql.org/October2021/#sec-Errors**. If an error occurs, the server response must have an **`errors`** field, containing a list of errors. An error must necessarily have the message field valued, which describes the type of error. Try running the **`findOrderByCode`** query for a non-existent order code, you will get a response similar to this:

```
{
  "errors": [
    {
```

```
      "message": "Order with code wrong-code not found",
      "locations": [],
      "extensions": {
        "classification": "NOT_FOUND"
      }
    }
  ],
  "data": {
    "findOrderByCode": null
  }
}
```

As you may have noticed, even with Spring for GraphQL, centralized error handling is easy to manage.

# Optimizing GraphQL with document parsing caching

Caching document parsing in GraphQL optimizes server performance by reducing redundant processing for repeated queries. Normally, each identical query requires parsing and validation, which are computationally intensive operations. By caching the parsed and validated document, the server can bypass these steps for recurring queries, enabling faster response times and more efficient resource utilization. In Spring for GraphQL, caching can be achieved by implementing a **`PreparsedDocumentProvider`** and registering it with **`GraphQlSourceBuilderCustomizer`**. In the **`GraphQLConfig`** class we add a bean of type **`GraphQlSourceBuilderCustomizer`** that uses an object of type **`PreparsedDocumentProvider`** to cache instances of **`Document`**:

```
@Configuration
@Slf4j
public class GraphQlConfig {

    ...

    @Bean
    public GraphQlSourceBuilderCustomizer sourceBuilderCustomizer() {
        return (builder) -> builder
                .configureGraphQl(graphQLBuilder -> graphQLBuilder
```

```
    }

    private PreparsedDocumentProvider preparsedDocumentProvider() {
        var cache = new ConcurrentHashMap<String, PreparsedDocumentEntry>();

        return (executionInput, parseAndValidateFunction) -> {
            if(cache.containsKey(executionInput.getQuery())) {
                log.info("ParsedDocumentEntry from cache");
                return cache.get(executionInput.getQuery());
            }
            else {
                var entry = parseAndValidateFunction.apply(executionInput);
                cache.put(executionInput.getQuery(), entry);
                return entry;
            }
        };

    }
}
```

In this example, we used a simple **`ConcurrentHashMap`**, but you can easily switch configuration to use a more complex cache.

To use the cache properly, we need to use variables in queries. So, the following query that does not use variables:

```
query orderQuery{
  findOrderByCode(orderCode: "5ZPK3173045768968A") {
    status
    totalPrice
    orderCode
    products {
      productCode
    }
  }
}
```

Can be transformed in this way, using the variable **`orderCode`**:

```
query orderQuery($orderCode: ID!){
  findOrderByCode(orderCode: $orderCode) {
    status
    totalPrice
    orderCode
    products {
      productCode
    }
  }
}
```

Declaring the variable **`orderCode`** like this:

```
{
  "orderCode": "5ZPK3173045768968A"
}
```

*Figure 4.5* shows a screenshot of the GraphQL query executed through the GUI provided by Spring:

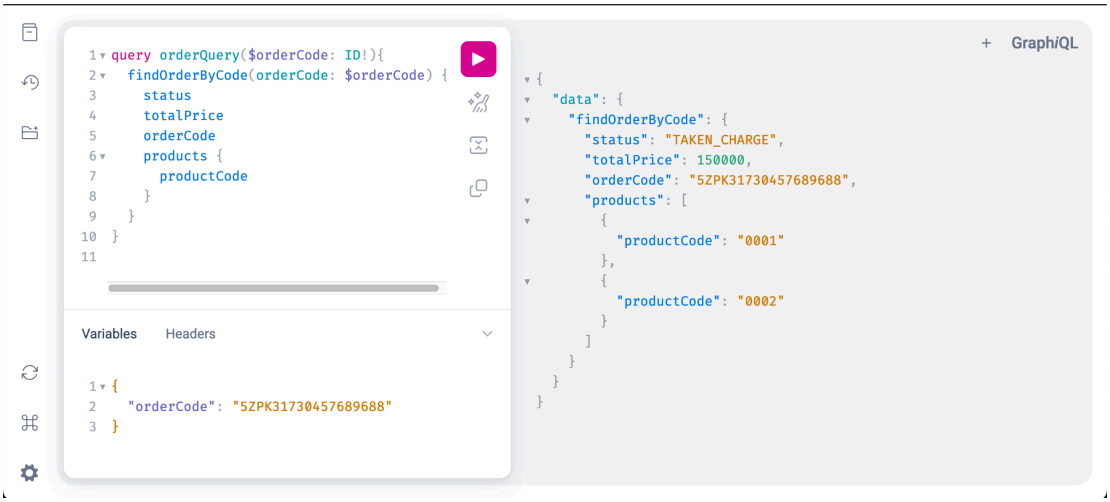



***Figure 4.5**: GraphQL query that uses a variable*

If you try to run the same query more than once, even changing the value of the variable, you will find the logging "ParsedDocumentEntry from cache" which indicates that the parsing was retrieved from the cache, optimizing query performance.

# GraphQL client in Spring

Spring for GraphQL provides the GraphQLClient client for making requests to a GraphQL server. The client is independent of the transport protocol used. You can create a **`GraphQLClient`** instance from the builder methods of the **`HttpGraphQlClient`**, **`WebSocketGraphQlClient`**, and **`RSocketGraphQlClient`** interfaces which use the HTTP, websocket, and rsocket protocols, respectively. During the time this chapter was written, the Spring for GraphQL release was at version 1.2.6: this version does not have a GraphQL HTTP client that supports RestClient, so we are forced to import the WebFlux dependency to use WebClient. However, version 1.3.0-SNAPSHOT already has a new interface, **`HttpSyncGraphQlClient`** which uses **`RestClient`**, so we will also have this interface available in future releases.

Let us try to create the graphql service microservice, a client that invokes the server API for the schema book, just for testing purposes. Add the Spring Boot WebFlux dependency to the project, as shown below:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Delete the spring-webflux dependency with the scope test that was automatically imported from Spring Initializr, otherwise, you will not be able to use WebClient. In the config package, create the **`GraphQlClientConfig`** class, annotated with **`@Configuration`**, and add the GraphQL client configuration:

```
@Bean
HttpGraphQlClient httpGraphQlClient() {
    return HttpGraphQlClient.builder()
            .url("http://localhost:8082/graphql")
            .build();
}
```

After that, in the same class, add a bean of type **`ApplicationRunner`**, so that it will run a query with the newly created client at application startup, as shown in the following code:

```
@Bean
ApplicationRunner applicationRunner(HttpGraphQlClient http) {
    return args -> {
        var query = """
          query bookDetails{
```

```
            bookById(id: "book-1") {
               id
               name
            }
        }
        """;
        http.document(query)
                .retrieve("bookById")
                .toEntity(Book.class)
                .subscribe(book -> log.info("Book: {}", book));
    };
}
```

The document method takes as input the definition of a GraphQL request. There is also the **`documentByName`** variant, which reads the definition of a query from the file instead of from the variable. The retrieve method takes input as the path of the query (or mutation) to be executed, finally, the **`toEntity`** method decodes the result of the query into the specified Java class. Restart the application; you will see a printout of the query result in the console.

# Testing GraphQL APIs

In order to test the GraphQL API, Spring provides the GraphQLTester test client that performs queries mapped from the server handlers. To inject an instance of GraphQLTester, the test class must be annotated with **`@GraphQlTest`**. Create the **`api`** package in the test folder and write the following test class:

```
@GraphQlTest
@Import({GraphQlConfig.class, ErrorHandler.class})
class OrderControllerTest {

    @Autowired
    private GraphQlTester graphQlTester;

    @MockBean
    private OrderService orderService;

    @Test
    void findOrderByCodeOkTest() {
        var productSummary = Product.builder().productCode("01")
```

```
                .build();

        var productDetail = Product.builder().productCode("01").
name("aProduct")
                .build();

        var order = Order.builder().orderCode("0001").status("DELIVERED")
                .products(List.of(productSummary))
                .build();

        var expectedOrder = order.toBuilder()
                .products(List.of(productDetail))
                .build();

        when(orderService.findOrderByCode("0001")).thenReturn(order);
        when(orderService.getProductsFromOrders(List.of(order)))
                .thenReturn(Map.of(order, List.of(productDetail)));

        graphQlTester
                .documentName("orderQueries")
                .operationName("orderDetails")
                .variable("id", "0001")
                .execute()
                .path("findOrderByCode")
                .entity(Order.class)
                .isEqualTo(expectedOrder);
    }
}
```

In addition to the **`@GraphQlTest`** annotation, we imported **`GraphQlConfig`** configurations to have custom scalars and error handling available. **`GraphQlTester`** queries the **`orderDetails`** operation declared in the **`orderQueries.graphql`** file, after which it verifies that the result of it is equal to the expected value. If not specified, by default **`GraphQlTester`** will look for the **`orderQueries.graphql`** file in the **`graphql-test/`** classpath. Then, create the **`graphql-test`** folder inside the test **`resources`** folder and write the following file **`orderQueries.graphql`**:

```
query orderDetails($id: ID){
    findOrderByCode(orderCode: $id) {
        orderCode
```

```
        status
        products {
            productCode
            name
        }
    }
}
```

Now, try running the test, you will see that it will yield a positive result. You can also easily test the case of providing a non-existent order code in the following way:

```
@Test
void findOrderByCodeKoTest() {
    var ex = new OrderNotFoundException("0001");
    when(orderService.findOrderByCode("0001"))
            .thenThrow(ex);
    graphQlTester
            .documentName("orderQueries")
            .operationName("orderDetails")
            .variable("id", "0001")
            .execute()
            .errors()
            .expect(actualError ->
                    actualError.getErrorType().equals(ErrorType.NOT_FOUND)
                            &&
                           actualError.getMessage().equals(ex.getMessage())
            );
}
```

The testing approach in Spring for GraphQL is the same as that taken in Spring MVC and Spring WebFlux.

# Comparing REST and GraphQL

When it comes to designing APIs, REST and GraphQL are two of the most prominent approaches, each with its unique strengths and challenges. Understanding the distinctions between them is crucial for choosing the right solution for your specific needs. Below, we will outline some of the key differences between REST and GraphQL to help you better

- In REST, a server provides several endpoints, e.g., for each resource, a different endpoint is associated. In GraphQL, there is a single endpoint to execute queries and mutations. It is the request body that specifies which query or mutation to execute.
- In REST, CRUD operations are identified by HTTP methods (GET, POST, PUT, DELETE). In GraphQL there are only three operations, query, mutation, and subscription. These operations are specified in the body of the request, using the POST method.
- In REST, to request data from different resources, the client must make multiple HTTP requests (under-fetching). In GraphQL, the client makes only one request to retrieve all the requested data.
- REST is also prone to retrieving more data than the client needs (over-fetching). In fact, even if the client is interested in a subset of the fields in the response, the server will always send the full response, following the provided OpenAPI specification. In GraphQL, only the fields requested by the client are returned to the client, optimizing the network bandwidth used.
- In REST, API versioning is important to handle the change of a schema, even for the addition of a simple field. In GraphQL, versioning is not necessary, as only the fields it requests are returned to the client, making the addition of any new schema fields transparent.

We also saw that REST and GraphQL share common features, for example, both are independent of the transport protocol used.

# Conclusion

In this chapter, you learned about the most important features of GraphQL, a modern query language for requesting and managing data from a server. The server can retrieve data from different resources, transparently to the client. It also allows optimizing data fetching based on what the client requests. You have seen how the Spring for GraphQL module implements GraphQL elegantly through annotations. You may be wondering in which cases to use REST and which cases to use GraphQL. As always, it depends on the specifics of the project. In general, if you need a great deal of flexibility and customization in data requests or manage an API that will evolve quickly and frequently, GraphQL may be the right choice. It may also be a good choice if your clients are frontends that need to retrieve data from different sources. For example, a scenario when we have a **back for front** (**BFF**) that fetching data from multiple servers and applications, GraphQL is the better choice. On the other hand, for applications with more stable and rigid requirements, or where caching and simplicity are priorities, REST may be more appropriate. For example, applications like blogs and e-commerce that require basic data retrieval and modifications and that do not require complex data handling and manipulation, REST is the better

choice. Anyway, the final decision will depend on the specifics of your project, the skills of the team, and the expectations of the API consumers.

# Points to remember

- Spring for GraphQL is independent of the transport protocol used. You therefore need to import the dependencies of Spring MVC, Spring WebFlux, Spring WebSocket, or Spring RSocket.
- GraphQL uses a single endpoint for all operations. In Spring for GraphQL, endpoint by default is /graphql. If you use the websocket protocol, the endpoint must be explicitly specified via the spring.graphql.websocket.path property.
- In GraphQL, only the requested data is returned to the client.
- You can eliminate the problem of N+1 requests by using the @BatchMapping annotation.
- Use @GraphQlTest and the GraphQlTester client to test the GraphQL API.

# Exercises

Although in this chapter we solved the problem of N+1 queries, if the client requests product fields that are not detail fields, e.g., it only requests the product code for each order, the catalog service HTTP API will still be invoked, even though the product code is already available. Try to optimize this by leveraging the DataFetchingFieldSelectionSet class that allows retrieving fields from the query made by the client and the GraphQLContext class that allows adding and retrieving context variables from different handler methods.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

**https://discord.bpbonline.com**



# Chapter 5
# Designing APIs with gRPC

## Introduction

In this chapter, we will discuss the modern landscape of API design using **Google Remote Procedure Call (gRPC)**, an open-source high-performance **Remote Procedure Call** (**RPC**) framework initially developed by *Google*. This chapter provides a thorough examination of gRPC and its integral components, offering insights into how it can transform the way applications communicate. We begin with an *Overview of gRPC* where we introduce the fundamental concepts of gRPC including its efficient use of HTTP/2, its default use of Protocol Buffers for data serialization, and how these features contribute to creating high-performance, scalable APIs that are language agnostic. This section aims to establish a solid understanding of why gRPC offers a compelling alternative to traditional REST APIs, particularly for microservices architectures where efficient communication between services is crucial. Following the overview, the chapter progresses to *Integrating gRPC into Spring Boot Applications*. Here, readers will learn step-by-step how to set up gRPC within a Spring Boot environment, effectively bridging the gap between theory and practice. This section includes detailed guidance on configuring server and client-side components, streamlining the development process, and demonstrating practical implementations that leverage the full capabilities of both Spring Boot and gRPC. Lastly, in *Testing gRPC APIs*, the chapter explores various strategies to ensure the reliability and robustness of gRPC services. By the end of this chapter, readers will not only understand the technical underpinnings of gRPC but also how to effectively implement, test, and deploy gRPC-

based APIs within a Spring Boot framework, ensuring they are well-prepared to use these techniques in their development projects.

# Structure

In this chapter, we will discuss the following topics:

- Overview of gRPC
- Integrating gRPC into Spring Boot applications
- Testing gRPC APIs

# Objectives

The primary aim of this chapter is to provide a comprehensive introduction to gRPC, its advantages, and its integration with the Spring Boot framework. The objectives are to educate readers on the powerful features of gRPC, including its high performance, scalability, bi-directional streaming, and compatibility with HTTP/2, as well as its efficient data serialization with Protocol Buffers. Additionally, this chapter aims to demystify the process of setting up gRPC within a Spring Boot application, guiding readers through practical steps and configurations necessary for successful integration. Moreover, an important goal is to arm developers with effective strategies and techniques for testing gRPC APIs, ensuring that they can build robust and reliable services. Through detailed explanations and practical examples, the chapter seeks to empower readers to design, implement, and maintain gRPC-based APIs that are optimized for modern microservices architectures. This will enable developers to harness the full potential of gRPC in their projects, improving the efficiency and performance of their systems.

# Overview of gRPC

gRPC is a modern open framework for RPC that Google initially developed. In gRPC, a client application can directly call a method of a server application located on a different machine as if it were invoking a local method. As in many RPC systems, gRPC is based on the idea of defining an interface called service where methods and parameters that can be invoked remotely are specified. These Service interfaces are defined using Protocol Buffers, providing a language-agnostic and platform-independent interface. The server implements the interface and starts a gRPC server to handle client requests. The client uses a stub that provides the same methods as the server interface. The stub acts as an intermediary between the client and the server, allowing the client to call methods on a remote server as if they were local methods.

Clients and servers may communicate in different environments and may be written in different languages. For example, you may have a gRPC server written in Java deployed on a Cloud environment and different clients written in other languages, such as *Python* and

*Go*, which are on on-premises machines. *Figure 5.1* shows an example of communication between a gRPC server with different gRPC clients:

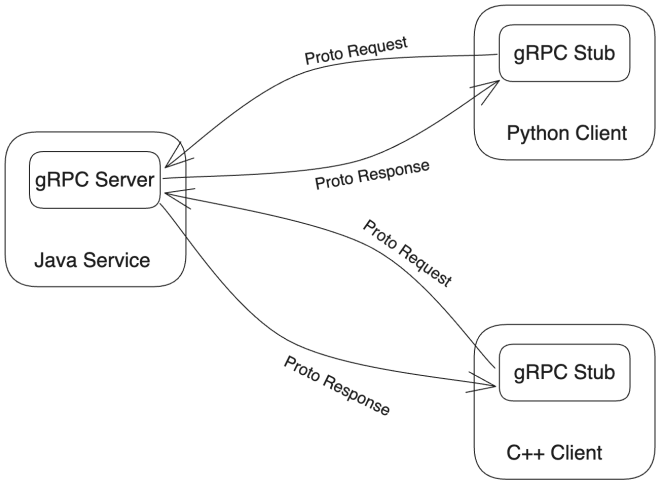



***Figure 5.1**: An example of communication between a gRPC server with different clients*

# The Protocol Buffers

By default, gRPC uses the Protocol Buffers format (also known as **protobuf**) as the interface language to define the data structure and serialization format. Protocol Buffers is an efficient binary serialization mechanism for defining how data should be structured in gRPC messages. Compared to JSON, which is the format used by default in REST, it is much more efficient in terms of size and serialization/deserialization speed.

When working with gRPC, the first step is to define the structure of the data in a proto file (file with the **`.proto`** extension). The data is structured as messages where each message is a small logical record of information containing a series of name-value pairs. Here is an example of a message:

```
message Person {
  string name = 1;
  int32 id = 2;
  bool has_ponycopter = 3;
}
```

Each field is associated with a unique numeric value which indicates the serialization order. Once the message is used, you cannot change the number associated with a field because it would correspond to deleting the field and creating a new one. You can rename a field without breaking client compatibility. You can also delete a field while maintaining client compatibility, then the client will receive the default value for that field (if the field

is of type string, the default value is an empty string; if it is a boolean, the default value is false, and so on).

In order to avoid incompatibilities when you delete a field, it is important to label, in the proto file, the field as **`reserved`**, so that the field number will not be associated with other fields in the future. In the label reserved, you can indicate either the field **`number`** or the field **`name`**; however, you cannot indicate both on the same label. An example of a valid use of reserved is as follows:

```
message Foo {
  reserved 2, 15, 9 to 11;
  reserved "foo", "bar";
}
```

Two other labels you can use are **`repeated`** and **`map`**. The repeated label is used to declare array or list-type fields that hold multiple values of the same data type, while the map is used to define fields similar to a hashmap or dictionary, storing data in a key-value format.

In the above example, the Person message has all fields of scalar types. However, you can also associate complex types and enumerations, as shown in the following example:

```
message SearchResponse {
  repeated Result results = 1;
  SearchType searchType = 2;
}

message Result {
  string url = 1;
  string title = 2;
}

enum SearchType {
  UNSPECIFIED = 1;
  BY_NAME = 2;
}
```

In addition to the message, services with their methods are also specified in the proto file, as shown by the following example:

```
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}
```

```
message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

Once you have completed the proto file, you can use the protocol buffer compiler, *protoc*, to generate the classes that map the messages and server and client service code in the build phase. *Figure 5.2* shows in detail the steps to generate the proto classes:

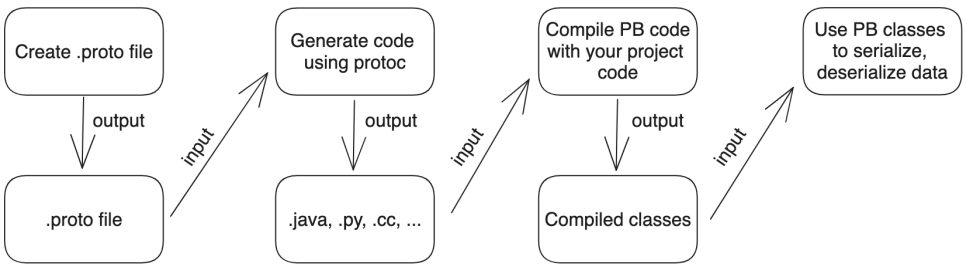



***Figure 5.2**: Steps to using proto classes*

# Communication in gRPC

gRPC uses HTTP/2 by default to transmit data. Compared with HTTP/1.1, the default version used by REST, HTTP/2 has features that make it more performant. For example, HTTP/2 introduces multiplexing of requests with the concept of streams. A stream is a collection of messages traveling over the same connection. A stream can be of short duration, for example, a GET call made by the client to receive timely information about a specific resource, or it could be of long duration, for example, the client wants to receive real-time updates on the status of a resource. Taking advantage of this feature, the following types of communication are possible in gRPC:

- **Unary RPC:** The simplest communication method, the client sends a single request via the stub method, and the server receives it and provides a single response to the client.
- **Server streaming RPC:** Like the previous method, instead of the server responding with a single message, it responds with a stream of messages. The connection ends when the server sends all the messages. This allows the server to transmit data as it becomes available.
- **Client streaming RPC:** This is similar to the Unary RPC method, except that the client does not send a single message as a request to the server but a stream of

messages. The server responds with a single message. Usually, but not necessarily, the server responds after it has received all the messages from the client.

- **Bidirectional streaming RPC:** Both client and server send a stream of messages. The logic of stream processing between client and server is application-specific. The client and server can read and write messages in any order because the two streams are independent. For example, the server could wait to receive all messages from the client before sending the message stream, or the server could send a message each time it receives one from the client (ping-pong logic).

The following is a proto file that uses all four communication types listed above:

```
service HelloService {
  rpc Unary(HelloRequest) returns (HelloResponse);
  rpc ServerStreaming(HelloRequest) returns (stream HelloResponse);
  rpc ClientStreaming(stream HelloRequest) returns (HelloResponse);
  rpc BidirStreaming(stream HelloRequest) returns (stream HelloResponse);
}

message HelloRequest {
  string greeting = 1;
}

message HelloResponse {
  string reply = 1;
}
```

# Comparison of gRPC and REST

From the preceding paragraphs, we can already get an idea about the differences between gRPC and REST; however, the following points aim to detail more about the differences between the two in terms of communication protocol, data format, performance, and use cases:

- **Communication protocol:**
    - **REST**: It uses HTTP/1.1 by default to transmit data. A connection can handle one request at a time.
    - **gRPC**: It uses HTTP/2 by default to transmit data. HTTP/2 introduces multiplexing, allowing a single TCP connection to handle several requests simultaneously. With this version of the HTTP protocol, gRPC enables faster and more efficient communication and exploits the concept of bidirectional streaming.

- **Data format:**
    - **REST**: By default, it uses JSON as its data format, which makes it both human and machine-readable. It is not bound to a schema file, so this format is more flexible than Protocol Buffers. However, it is less efficient in terms of size and parsing speed than binary formats.
    - **gRPC**: By default, it uses Protocol Buffers (protobuf) as its data format, which is a binary, compact, and fast serialization system. However, it needs a schema definition (proto file) that must be shared between the client and server.
- **Performance:**
    - **REST**: It may experience higher latencies due to the overhead of HTP/1.1 and JSON parsing, especially when handling large volumes of data and requiring high performance.
    - **gRPC**: It offers superior performance to REST using HTTP/2 and protobuf.

We published a performance comparison of gRPC and REST using the Apache Bench tool on a microservice that acts as a client to a gRPC server and a REST server. Both servers return the same payload. *Figure 5.3* shows the results of the comparison:

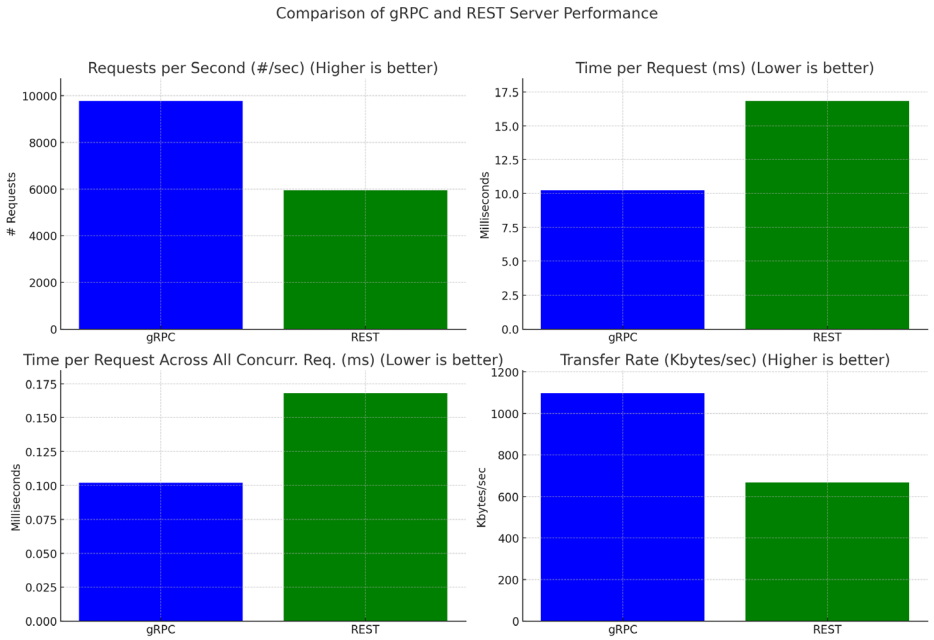



***Figure 5.3**: A graph comparing the performance of gRPC and REST*

As you can see, gRPC wins in all aspects. If you would like to learn more about this comparison, visit the link **https://github.com/vincenzo-racca/grpc-vs-rest-performance**.

The use cases are mentioned below:

- **REST:** It is extremely versatile; it can be used for all web applications. Documentation through OpenAPI makes it suitable for public APIs, whether consumed by browsers or other microservices. If high performance is not required, you may also prefer it for internal APIs because of its ease of implementation.
- **gRPC:** By default, it is more performant than REST. It is particularly suitable in internal communication between services in microservice architectures where performance and efficiency aspects are considered critical.

# Integrating gRPC into Spring Boot applications

Unlike REST and GraphQL, there is no Spring project developed by the Spring community dedicated to gRPC. However, since Spring is an open-source project, there are several open-source projects that integrate Spring Boot with gRPC, created by other developers. The most widely used is gRPC Spring Boot Starter (**https://github.com/grpc-ecosystem/grpc-spring**). This project includes both a client module and a server module.

In this chapter, we will not create a new microservice, but extend the catalog service and order service microservices so that they also act as gRPC server and gRPC client, respectively. We will create a new implementation of the ProductClient interface of order service so that it can also communicate with catalog service via gRPC, as shown in *Figure 5.4*:

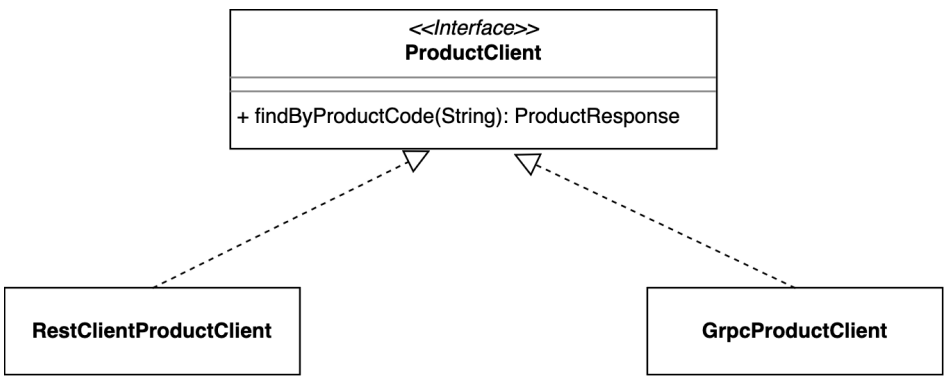



***Figure 5.4**: The class diagram of ProductClient with its REST and gRPC implementations*

This is a good use case of gRPC since an order might contain many products and this might impact order service performance during new order entry. With gRPC, we can optimize the `findProductByCode` call by utilizing Protocol Buffers for efficient serialization and maintaining a single persistent connection. By leveraging request multiplexing, multiple product lookups can be processed concurrently and efficiently within the same connection.

All the code is available in the **chapter-05** folder of the project's GitHub repository. The paragraph is divided into two subsections, one dedicated to implementing the gRPC server and one dedicated to implementing the gRPC client.

# Implementation of the gRPC server

In order to make the catalog service also act as a gRPC server, we import the server module dependency of gRPC Spring Boot Starter into the **pom.xml**, as shown below:

```
<dependency>
    <groupId>net.devh</groupId>
    <artifactId>grpc-server-spring-boot-starter</artifactId>
    <version>${grpc-spring-boot.version}</version>
</dependency>
```

However, the gRPC Spring Boot Starter project does not handle reactive flows. Since catalog service uses WebFlux, we do not want gRPC calls to be blocked. In order to use reactive flows in gRPC, we also need to import the reactive gRPC dependency developed by Salesforce, dedicated to Reactor, as shown in the following code:

```
<dependency>
    <groupId>com.salesforce.servicelibs</groupId>
    <artifactId>reactor-grpc-stub</artifactId>
    <version>${reactor-grpc.version}</version>
</dependency>
```

Finally, we need to import the protobuf plugin for Maven which allows Java classes to be generated from a proto file. Since Proto Buffers is an operating system-dependent binary format, we first include an extension that allows you to derive the type of operating system used during the build phase. Add the following piece of code to the build section of the **pom.xml** file:

```
<extensions>
    <extension>
        <groupId>kr.motd.maven</groupId>
        <artifactId>os-maven-plugin</artifactId>
        <version>1.7.1</version>
    </extension>
</extensions>
```

Now, we can add them in the plugins section, proto plugin. Refer to the following code for a better understanding:

```
<plugin>
    <groupId>org.xolstice.maven.plugins</groupId>
    <artifactId>protobuf-maven-plugin</artifactId>
    <version>0.6.1</version>
    <executions>
        <execution>
            <goals>
                <goal>compile</goal>
                <goal>compile-custom</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <protocArtifact>
            com.google.protobuf:protoc:3.25.1:exe:${os.detected.classifier}
        </protocArtifact>
        <pluginId>grpc-java</pluginId>
        <pluginArtifact>
            io.grpc:protoc-gen-grpc-java:1.63.0:exe:${os.detected.
classifier}
        </pluginArtifact>
        <protocPlugins>
            <protocPlugin>
                <id>reactor-grpc</id>
                <groupId>com.salesforce.servicelibs</groupId>
                <artifactId>reactor-grpc</artifactId>
                <version>${reactor-grpc.version}</version>
                <mainClass>
                    com.salesforce.reactorgrpc.ReactorGrpcGenerator
                </mainClass>
            </protocPlugin>
        </protocPlugins>
    </configuration>
</plugin>
```

In the properties section of the **pom.xml**, we add the versions of the dependencies we just imported, as shown below:

```
<reactor-grpc.version>1.2.4</reactor-grpc.version>
<grpc-spring-boot.version>3.1.0.RELEASE</grpc-spring-boot.version>
```

The second step is to create the proto file. The plugin will generate the Java classes from the proto files located inside the **src/main/proto** folder. Then, we will create the **proto** folder inside **src/main** and write the following **product.proto** file:

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "com.easyshop.catalogservice.proto";
option java_outer_classname = "ProductProto";

service ProductProtoService {
  rpc FindProductByCode(ProductProtoRequest) returns (ProductProtoRe-
sponse) {
  }
}

message ProductProtoRequest {
  string code = 1;
}

message ProductProtoResponse {
  string code = 1;
  string category = 2;
  int64 price = 3;
}
```

Let us analyze the file, refer to the following points for a better understanding:

- The **syntax = "proto3"** line indicates the version of the proto used. In this case, we use the latest version 3.
- The **java_multiple_files = true** line indicates that the messages within the proto file should be generated as separate **.java** files. By default, the option is set to false, then the **ProductProtoRequest** and **ProductProtoResponse** classes would be generated as static inner classes of the class indicated by the **java_outer_classname** option, i.e., ProductProto.

- The **`java_package = "com.easyshop.catalogservice.proto"`** line indicates the package where the Java classes that map messages are autogenerated. If not indicated, no package will be used.
- The **`java_outer_classname = "ProductProto"`** line allows you to specify the name of the outer class, which contains the descriptors of the classes that map messages. In addition, if the **`java_multiple_files`** option is set to false, it also contains the classes that map messages. If the **`java_outer_classname`** option is not made explicit, by default, the name of the outer class will be the same as the name of the proto file, in this case, Product.

Finally, the proto file contains the **`ProductProtoService`** service that has the **`FindProductByCode`** method which receives a message of type **`ProductProtoRequest`** as input and returns a message of type **`ProductProtoResponse`**.

In order to generate the classes, use the usual **`./mvnw clean compile`** command from the root of the catalog service project. After running the command, you will find the autogenerated classes in the **`target/generated-sources/protobuf`** folder, as shown in *Figure 5.5*:

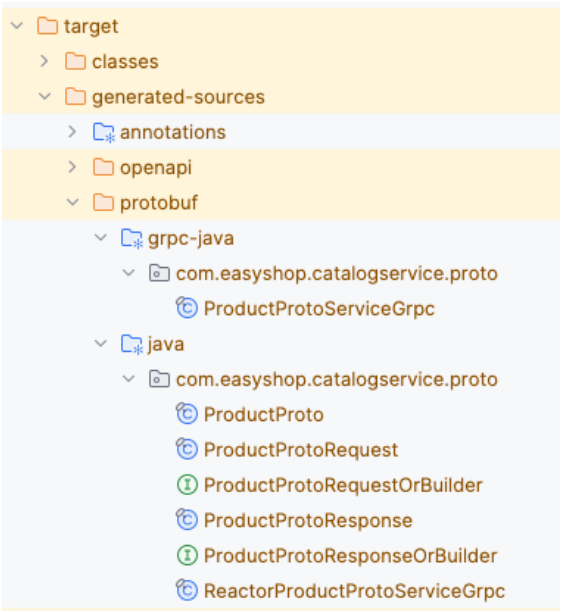



***Figure 5.5**: The proto classes autogenerated by the Maven plugin*

Now, we can implement the **`proto-file`** service. This class, in addition to implementing the **`findProductByCode`** method of the autogenerated **`ProductProtoServiceImplBase`** class, will need to be annotated with **`@GrpcService`** which marks the class as a gRPC service and as a Spring bean. In addition, the annotation allows interceptors to be accepted among its parameters, which is useful for intercepting service requests and responses. Create the

package **grpc.service** and write the class that implements the gRPC service, as shown below:

```
@GrpcService
@RequiredArgsConstructor
@Slf4j
public class ProductProtoServiceGrpcImpl extends
        ReactorProductProtoServiceGrpc.ProductProtoServiceImplBase {

    private final ProductService productService;


    @Override
    public Mono<ProductProtoResponse> findProductByCode(
            Mono<ProductProtoRequest> request) {
        return request
                .doOnNext(protoRequest ->
                        log.info("Proto request: {}", protoRequest))
                .flatMap(protoRequest ->
                        productService.findProductByCode(protoRequest.getCode()))
                .map(productResponse -> ProductProtoResponse.newBuilder()
                        .setCode(productResponse.getCode())
                        .setCategory(productResponse.getCategory().getValue())
                        .setPrice(productResponse.getPrice())
                        .build())
                .doOnNext(protoResponse ->
                        log.info("Proto response: {}", protoResponse))
                .doOnError(ex ->
                        log.error("Error in findProductByCode with request: {}"
                                , request, ex));
    }
}
```

The method has no logic, it simply calls the method of the **ProductService** class, which handles the business logic, and then maps the **ProductService** response into an object of type **ProductProtoResponse**.

However, **ProductService** throws an exception of type **ProductNotFoundException** if the product with the given input code does not exist. This exception is already handled

by our **`ErrorHandler`**, but the latter only handles exceptions for REST requests; it would not work for gRPC. In order to centralize gRPC API errors, we can still use the same approach used for REST while using different annotations, i.e., we can annotate with **`@GrpcAdvice`**, the class that acts as the error handler, and we can annotate its methods with **`@GrpcExceptionHandler`**. Create the exception subpackage within the **`grpc`** package and write a class that acts as an error handler for the gRPC service you just created, as shown below:

```
@GrpcAdvice
public class ErrorProtoHandler {


    @GrpcExceptionHandler(ProductNotFoundException.class)
    public StatusRuntimeException handle(ProductNotFoundException ex) {
        var status = Status.NOT_FOUND
                .withDescription(ex.getMessage())
                .withCause(ex);
        return status.asRuntimeException();
    }
}
```

The **`GrpcExceptionHandler`** annotation can define one or more exception types. In such cases, only the specified types will be handled by the annotated method. The method's parameters must be compatible with the specified exception types, meaning that each exception type in the annotation must match or be a superclass of at least one of the method's parameters.

The method of the handler returns an object of type **`StatusRuntimeException`** which allows the API's status information to be propagated through an exception of type **`RuntimeException`**. Status in gRPC is like the concept of HTTP status for REST. Methods of a class annotated with **`@GrpcAdvice`** must necessarily return one of the following object types: **`Status`**, **`StatusException`**, **`StatusRuntimeException`**, or **`Throwable`**.

We start the catalog service application after starting the Postgres container. The gRPC server by default is started on port 9090. You can change this configuration by setting the property **`grpc.server.port`** in the **`application.properties`** file.

In order to test the service, we can use a tool like cURL but dedicated to gRPC called **grpcurl**. You can download grpcurl for *Mac*, *Windows*, or *Linux*. Take a look at the links **https://github.com/fullstorydev/grpcurl?tab=readme-ov-file#installation** and **https://github.com/fullstorydev/grpcurl/releases** for a better understanding.

If the product table is empty, enter two products with codes 0001 and 0002 using the catalog service POST API, as shown below:

```
http :8080/products code=0001 name="ALaptop" category=lap-
top price=60000 brand=FirstBrand
http :8080/products code=0002 name="SuperLaptop" category=lap-
top price=30000 brand=FirstBrand
```

We now use **grpcurl** to retrieve the product with code 0001, as shown below:

```
grpcurl -d ‹{"code": "0001"}’ -plaintext \
  localhost:9090 ProductProtoService.FindProductByCode
{
  "code": "0001",
  "category": "laptop",
  "price": "60000"
}
```

As shown by the code above, the product with code 0001 is returned correctly. Now, let us try to request a product with a non-existent code:

```
grpcurl -d ‘{"code": "0005"}’ -plaintext \
  localhost:9090 ProductProtoService.FindProductByCode
ERROR:
  Code: NotFound
  Message: Product with code 0005 not found
```

As we expected, the **NotFound** error is returned with the message that the product was not found. Our gRPC server is working fine, we just need to implement the gRPC client.

# Implementation of the gRPC client

In order to make the order service act as gRPC client to catalog service, we import the client module dependency of gRPC Spring Boot Starter into the **pom.xml**, as depicted below:

```
<dependency>
    <groupId>net.devh</groupId>
    <artifactId>grpc-client-spring-boot-starter</artifactId>
    <version>${grpc-spring-boot.version}</version>
</dependency>
```

Also, since this module uses Java Validation annotations with the old **javax** package, we are forced to import the following dependency as well:

```
<dependency>
    <groupId>javax.annotation</groupId>
```

```
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```

In the catalog service, we did not need to import this dependency because it is already included in the reactive gRPC module. Add in the plugins section of the **pom.xml**, the plugin that generates the proto classes. Refer to the following code for a better understanding:

```
<plugin>
    <groupId>org.xolstice.maven.plugins</groupId>
    <artifactId>protobuf-maven-plugin</artifactId>
    <version>0.6.1</version>
    <executions>
        <execution>
            <goals>
                <goal>compile</goal>
                <goal>compile-custom</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <protocArtifact>
            com.google.protobuf:protoc:3.25.1:exe:${os.detected.classifier}
        </protocArtifact>
        <pluginId>grpc-java</pluginId>
        <pluginArtifact>
            io.grpc:protoc-gen-grpc-java:1.63.0:exe:${os.detected.
classifier}
        </pluginArtifact>
    </configuration>
</plugin>
```

Finally, add the Starter gRPC dependency version to the properties section of the **pom.xml**:

```
<grpc-spring-boot.version>3.1.0.RELEASE</grpc-spring-boot.version>
```

Copy the **proto** folder of the catalog service and paste it into the **src/main** folder of the order service, then generate the proto classes with the usual Maven command.

Now, we can create the **GrpcProductClient** class that implements **ProductClient** and

uses the stub method of the **`ProductProtoService`** service. You can inject the gRPC client into the Spring context using the **`@GrpcClient`** annotation. This annotation accepts the name of the service in the parameters since the client application could use multiple gRPC services. In the **`middleware.msclient.impl`** package, write the **`GrpcProductClient`** class, as shown below:

```
@Component
@Slf4j
public class GrpcProductClient implements ProductClient {

    private final ProductProtoServiceGrpc.ProductProtoServiceBlockingStub
            productProtoService;

    public GrpcProductClient(@GrpcClient("productProtoService")
                             ProductProtoServiceGrpc.
ProductProtoServiceBlockingStub
                                     productProtoService) {
        this.productProtoService = productProtoService;
    }

    @Override
    public ProductResponse findByProductCode(String productCode) {
        var request = ProductProtoRequest.newBuilder()
                .setCode(productCode)
                .build();
        try {
            var product = productProtoService.findProductByCode(request);
            log.debug("Product with code {} found", product.getCode());
            return new ProductResponse(
                    product.getCode(),
                    product.getCategory(),
                    product.getPrice()
            );
        }
        catch (StatusRuntimeException ex) {
            if(Status.NOT_FOUND.getCode().equals(ex.getStatus().
```

```
getCode())) {
                throw new ProductOrderNotFoundException(productCode);
            }
            throw ex;
        }
    }
}
```

The gRPC Client instance of type **`ProductProtoServiceBlockingStub`** (an autogenerated class from the proto plugin) is injected into the constructor with the **`@GrpcClient`** annotation. The **`findByProductCode`** method simply calls the gRPC client method of the same name.

We make it so that you can choose to use the gRPC client or the REST client depending on a property called **`order.service.product-client-type`**. If this property equals grpc, the gRPC client will be used, if it equals rest, the REST client will be used. In order to do that, Spring provides the **`@ConditionalOnProperty`** annotation. We add this annotation on the **`GrpcProductClient`** class, as shown below:

```
@ConditionalOnProperty(name = "order.service.product-client-
type", havingValue = "grpc")
```

We also add the same annotation on the **`RestClientProductClient`** class, as depicted below:

```
@ConditionalOnProperty(name = "order.service.product-client-
type", havingValue = "rest")
```

The **`order.service.product-client-type`** property is managed, like all other properties, by the **`OrderServiceProperties`** class. We then add the **`productClientType`** field to the class as follows:

```
@Component
@ConfigurationProperties(prefix = "order.service")
@Data
public class OrderServiceProperties {

    private String catalogserviceUrl;
    private ProductClientType productClientType;

    public enum ProductClientType {
        REST,
        GRPC
    }
```

```
}
```

All that remains is to set the property in the **`application.properties`** file. We chose to use gRPC by default, giving the value of propery as follows:

```
order.service.product-client-type=grpc
```

Also, in the **`application.properties`** file, we need to add the gRPC server endpoint tied to the service indicated in the **`@GrpcClient`** annotation:

```
grpc.client.productProtoService.address=static://localhost:9090
grpc.client.productProtoService.negotiation-type=plaintext
```

We set the property **`grpc.client.productProtoService.negotiation-type=plaintext`** since the default value is **`tls`**. Since the remote endpoint does not use a **Secure Sockets Layer** (**SSL**) connection, setting it to plaintext disables SSL/TLS negotiations and allows communication over plain text, as our server is not configured for secure communication.

Start order service and try to create an order with existing product codes, as shown below:

```
http :8081/orders \
  Content-Type:application/json \
  "products[0][productCode]"=0001 \
  "products[0][quantity]"=2 \
  "products[1][productCode]"=0002 \
  "products[1][quantity]"=1
```

The response will look like the code given below:

```
HTTP/1.1 201
Location: http://localhost:8081/orders/DIBCJ1714497559268
```

This means that the gRPC communication was successful. Try creating an order with a non-existing product code. The response will look like the code given below:

```
HTTP/1.1 400
{
    "detail": "Product with code 0005 not found",
    "instance": "/orders",
    "status": 400,
    "title": "Bad Request",
    "type": "about:blank"
}
```

The case of a product with non-existing code is handled correctly. We have made it transparent to order service clients that the microservice communicates with catalog service via gRPC and no longer REST while gaining efficiency and speed.

# Testing gRPC APIs

In order to test the gRPC server with jUnit, we can write integration tests, simulating that a gRPC client makes calls to the server.

In catalog service, add the client module dependency of gRPC, with scope test, as shown below:

```
<dependency>
    <groupId>net.devh</groupId>
    <artifactId>grpc-client-spring-boot-starter</artifactId>
    <version>${grpc-spring-boot.version}</version>
    <scope>test</scope>
</dependency>
```

The gRPC module provides an in-process server that allows a gRPC server to run in the same process as the client, this is useful for the testing phase. After that, we can create a test that instantiates a gRPC client that calls the gRPC in-process server. In the test folder, create the package **grpc.service** and write the following class:

```
@SpringBootTest(properties = {
        "grpc.server.in-process-name=test", // Enable inProcess server
        "grpc.server.port=-1", // Disable external server
        "grpc.client.inProcess.address=in-process:test"
})
class ProductProtoServiceGrpcImplTest {

    @GrpcClient("inProcess")
    private ReactorProductProtoServiceGrpc.ReactorProductProtoServiceStub
            protoServiceStub;

    @MockBean
    private ProductService productService;

    @Test
    void findProductByCodeOkTest() {
```

```
        var productCode = "0001";
        var category = ProductCategory.LAPTOP;
        var price = 50000L;
        var request = ProductProtoRequest.newBuilder()
                .setCode(productCode)
                .build();

        var expectedResponse = ProductProtoResponse.newBuilder()
                .setCode(productCode)
                .setCategory(category.getValue())
                .setPrice(price)
                .build();

        when(productService.findProductByCode(productCode))
                .thenReturn(Mono.just(new ProductResponse()
                        .code(productCode)
                        .category(category)
                        .price(price)));

        StepVerifier.create(protoServiceStub.findProductByCode(request))
                .expectNext(expectedResponse)
                .verifyComplete();
    }
}
```

The test is similar to the one performed for the REST API. The service call is mocked, after which the gRPC client is used to make the call to the server and verify that the result received is the same as the expected result. To test a KO case, just change the expected result of the service. We leave it up to you to write this test, you can still find it in the project's GitHub repository.

In order to test the gRPC client in order service, we can use a library that mocks the gRPC server or we can create a mock class that acts as a gRPC server. In this example, we chose the second way.

First, in order service, import the gRPC server module with scope test, as shown below:

```
<dependency>
    <groupId>net.devh</groupId>
    <artifactId>grpc-server-spring-boot-starter</artifactId>
```

```
    <version>${grpc-spring-boot.version}</version>
    <scope>test</scope>
</dependency>
```

Then, in the test folder, create the config package and write the mock class that acts as the gRPC server. Refer to the following code for a better understanding:

```
@GrpcService
public class ProductProtoServiceGrpcMock
        extends ProductProtoServiceGrpc.ProductProtoServiceImplBase {

    @Override
    public void findProductByCode(ProductProtoRequest request,
                          StreamObserver<ProductProtoResponse> responseOb-
server) {

        if(request.getCode().equals("not-found-code")) {
            var status = Status.NOT_FOUND.withDescription("Product not Found");
            responseObserver.onError(status.asException());
        }
        else {
            var response = ProductProtoResponse.newBuilder()
                    .setCode(request.getCode())
                    .setCategory("laptop")
                    .setPrice(50000L)
                    .build();

            responseObserver.onNext(response);
            responseObserver.onCompleted();
        }
    }

}
```

When the server receives a *not-found* product code as input, it returns **NOT_FOUND** status to the client, while in all other cases, it always returns a valid response. In the same package, let us create the Spring configuration for our test so that we can start the context for gRPC using the newly created gRPC server:

```
@Configuration
```

```
@ImportAutoConfiguration({
        GrpcServerAutoConfiguration.class, // Create required server beans
        GrpcServerFactoryAutoConfiguration.class, // Select server implementation
        GrpcClientAutoConfiguration.class}) // Support @GrpcClient annotation
public class GrpcTestConfig {

    @Bean
    ProductProtoServiceGrpcMock productProtoServiceGrpcMock() {
        return new ProductProtoServiceGrpcMock();
    }

    @Bean
    ProductClient productClient(@GrpcClient("productProtoService")
                                ProductProtoServiceGrpc.
                                        ProductProtoServiceBlockingStub
                                        productProtoService) {

        return new GrpcProductClient(productProtoService);
    }

}
```

We can use this configuration in our test to verify that the gRPC client is working properly. Again, we use an in-process server. In the test folder, create the **middleware.msclient** package and write the following test class:

```
@SpringBootTest(properties = {
        "grpc.server.inProcessName=test", // Enable inProcess server
        "grpc.server.port=-1", // Disable external server
        "grpc.client.productProtoService.address=in-process:test"
})
@SpringJUnitConfig(classes = { GrpcTestConfig.class })
class GrpcProductClientTest {

    @Autowired
    private ProductClient productClient;

    @Test
    void validateProductCodeOkTest() {
```

```
        var actualResponse = productClient.findByProductCode("0001");
        assertThat(actualResponse).isEqualTo(expectedResponse);
    }

    @Test
    void validateProductCodeKoTest() {
        assertThatExceptionOfType(ProductOrderNotFoundException.class)
                .isThrownBy(() ->
                        productClient.findByProductCode("not-found-code"));
    }
}
```

The two test methods verify the correct operation of the client in the case of an existing product code and a non-existing product code.

Note that the order service **ProductClientTest** class tests that we wrote in *Chapter 3, Easily Scalable APIs with Virtual Thread*, will no longer work because the test class context will take the configuration of the **order.service.product-client-type=grpc** property set previously in the **application.properties** file. To get around this problem, you can create the file **application-test-rest.properties** in the test resources folder, which contains the following content:

```
order.service.product-client-type=rest
```

Spring allows us to associate **application.properties** files automatically with Spring profiles, using this naming-convention: **application-{profile}.properties**. By using the **@ActiveProfiles("test-rest")** annotation in the **ProductClient** class, we can specify which application context to load with the profile. For example, if we are testing with a local configuration, we can set it to local, and similarly, for UAT or production configurations, we can set it to uat or prod. This flexibility ensures that the right configuration is loaded for the appropriate environment. Try relaunching the tests in the class; they will work correctly again.

Using the gRPC Spring Boot Starter project, we were able to easily test both the client and server code of gRPC.

# Conclusion

In this chapter, we saw how gRPC can revolutionize API design, particularly in modern contexts such as microservice architectures and distributed applications. Through a detailed exploration of gRPC's features and capabilities, we learned how this framework, based on HTTP/2 and Protocol Buffers, can provide significantly more efficient and faster inter-service communication than traditional REST-based approaches. The gRPC

framework using HTTP/2, improves latency and throughput through multiplexing and header compression. It also facilitates the development of robust APIs with its strong typing and interface definition system through proto files. This allows development teams to generate code for clients and servers in different programming languages automatically, reducing errors and improving development efficiency. We have seen that although there is currently no Spring module for gRPC created by the Spring community, the integration of gRPC with Spring Boot is easy due to the gRPC Spring Boot Starter project. This project includes both the server and client modules, so you can easily use it for both scenarios. In addition, you can use the same project to run server-side and client-side integration tests. In summary, adopting gRPC can mean a significant step forward for organizations seeking to optimize their backend architectures, especially those that require intensive and frequent communications between microservices. With its ability to efficiently handle complex data flows and high workloads, gRPC is positioned as a strategic choice for the future of API design. So far, we have delved into different synchronous communication techniques with REST, GraphQL, and gRPC. In the next chapter, *Aynchronous APIs with Spring Cloud Stream and Apache Kafka*, we will see how to apply asynchronous communication with Spring Cloud Stream and Apache Kafka.

# Points to remember

- By default, gRPC uses HTTP/2, which compared to HTTP/1.1 offers more advanced features with multiplexing and header compression, making it more efficient and faster.
- By default, gRPC uses Protocol Buffers (protobuf) as its data format. This is a binary format that allows it to be more efficient than the JSON format.
- The server and client code of gRPC is self-generated from the proto files.

# Exercises

Try replacing a few REST calls with gRPC in the graphl service microservice so that the performance of the latter is optimized as well.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

**https://discord.bpbonline.com**



# Chapter 6
# Asynchronous APIs with Spring Cloud Stream and Apache Kafka

## Introduction

In this chapter, we will explore the dynamic world of asynchronous communication within microservices architectures. This chapter delves into the foundational concepts and practical implementations of asynchronous **Application Programming Interfaces** (**APIs**) using two powerful technologies: Apache Kafka and Spring Cloud Stream. Designed to equip developers with the knowledge to build resilient, scalable, and efficient message-driven systems, this chapter offers a comprehensive guide through the complexities and capabilities of modern asynchronous messaging frameworks.

The chapter begins with an exploration of *Asynchronous APIs*, discussing the advantages of asynchronous communication patterns over traditional synchronous models. Further sections highlight how asynchronous methods can enhance scalability, improve resource utilization, and provide better handling of long-running operations or high-latency tasks.

Following the introduction to asynchronous principles, we will understand *Apache Kafka*, a renowned distributed event streaming platform known for its high throughput, reliability, and scalability. Here, we will cover the architecture of Kafka, its core components, and how it supports robust message brokering capabilities that are critical for the processing of large streams of data in real-time.

Next, we introduce *The Spring Cloud Stream Project*, an innovative framework designed to simplify the development of message-driven microservices. This section outlines how

Spring Cloud Stream abstracts away many of the complexities associated with standard message protocols and infrastructures, providing a unified model for building message-driven applications.

# Structure

In this chapter, we will discuss the following topics:

- Asynchronous APIs
- Introduction to Apache Kafka
- Introduction to Spring for Apache Kafka
- The Spring Cloud Stream Project
- Implementing message-driven microservices with Spring Cloud Stream
- Error handling in Spring Cloud Stream
- Testing Spring Cloud Stream applications

# Objectives

The primary objective of this chapter is to impart a solid understanding of the architecture and functionality of asynchronous APIs, focusing on their implementation using Apache Kafka and Spring Cloud Stream. This exploration aims to familiarize readers with the fundamental concepts and advantages of asynchronous communication, which is critical for building scalable and efficient microservices architectures.

A key goal is to introduce Apache Kafka, elucidating its robust architecture and how it can be utilized to handle massive volumes of real-time data transactions. Through this, readers will gain insights into Kafka's design principles, its reliability, and its scalability features, which are essential for developing high-performance applications.

Furthermore, this chapter seeks to introduce and expound on the Spring Cloud Stream framework. The objective here is to demonstrate how this framework simplifies the development of message-driven microservices by abstracting complex boilerplate configurations and providing a streamlined approach to building, deploying, and operating event-driven systems.

Another critical aim is to guide readers through the practical implementation of message-driven microservices using Spring Cloud Stream. This includes providing comprehensive examples and step-by-step instructions on setting up and managing data flows, which will help readers effectively apply their knowledge in real-world scenarios.

Additionally, the chapter focuses on robust error-handling mechanisms within Spring Cloud Stream applications. Understanding how to manage errors and exceptions is paramount to ensuring the reliability and resilience of applications, especially in a

Lastly, the chapter aims to cover the testing of Spring Cloud Stream applications. Effective testing strategies are vital for verifying the functionality and performance of asynchronous systems, ensuring they meet the required standards and behave as expected under various conditions.

By achieving these objectives, the chapter will equip readers with the necessary skills and knowledge to design, implement, and maintain sophisticated asynchronous API systems using Apache Kafka and Spring Cloud Stream, thereby enhancing their capability to develop advanced and scalable microservices architectures.

# Asynchronous APIs

In the *first chapter*, *Introduction to REST Architecture and API-first Approach*, we introduced the concept of asynchronous communication between microservices and compared it with synchronous communication. Asynchronous communication is useful for services that need to receive real-time updates, in cases where a response from the server cannot be received quickly, or when the server does not immediately have data available to return to the client. In this section, we will explore some different implementations of asynchronous communication.

## HTTP polling

HTTP polling is not a type of asynchronous communication, however, it is a technique that allows a client to query the server multiple times, via HTTP requests, to get continuous data updates. However, with this technique, you will not have real-time updates and more importantly it can generate a high amount of unnecessary traffic. This technique is also called short polling. The diagram in *Figure 6.1* summarizes how short polling works:

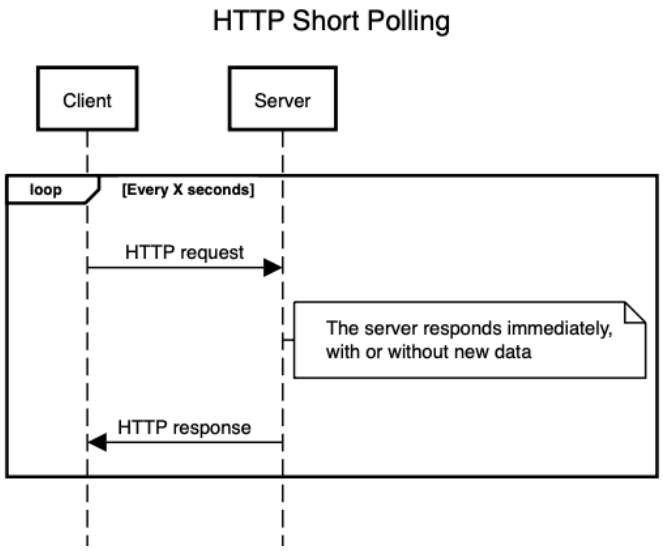



***Figure 6.1***

There is a more advanced polling technique called long polling, where the client sends a request to the server; the server keeps the request open until it has new data to send or until a timeout expires. Compared with short polling, this technique has the advantage of reducing unnecessary network traffic because the client makes fewer requests to the server. However, this technique can keep many connections open on the server, which could consume significant resources if the number of clients is large. The diagram in *Figure 6.2* shows how long polling works:

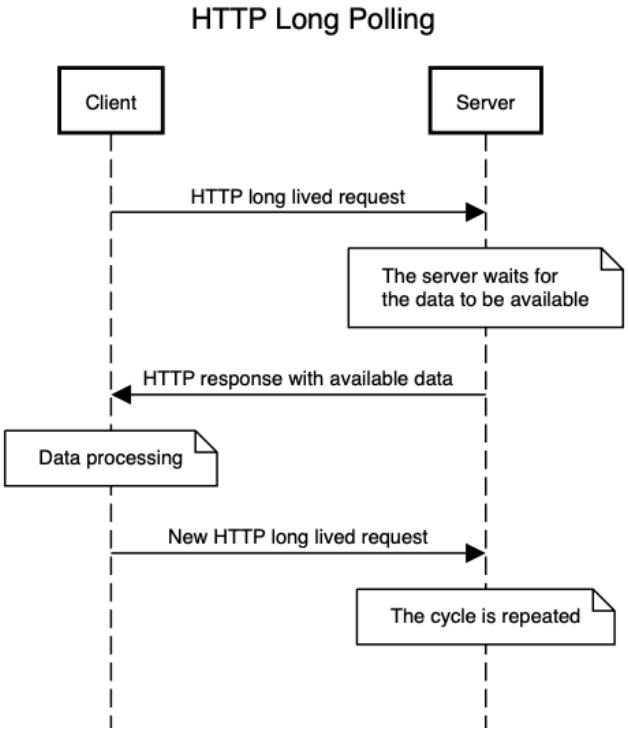



***Figure 6.2**: Operation of HTTP long polling*

HTTP polling is a simple technique to implement but is often not the most efficient solution for applications requiring real-time updates or handling large numbers of clients. It is used in applications where implementation of more advanced solutions such as WebSocket or **Server-Sent Events** (**SSE**) is considered excessive in complexity and a simpler and faster, albeit inefficient, implementation is preferred.

# Webhook

Webhooks are a technique used to allow an application to send real-time data to other applications over HTTP. Unlike polling, the server sends data to clients as soon as it is available. The way it works is as follows: the server provides a registration API, where the client specifies as input a callback URL and any event types on which it is interested

in receiving updates. The client then performs the registration. When a specific event occurs, the server notifies the client by invoking the callback URL provided to it during registration, using the POST method. The diagram in *Figure 6.3* summarizes the operation of Webhooks:

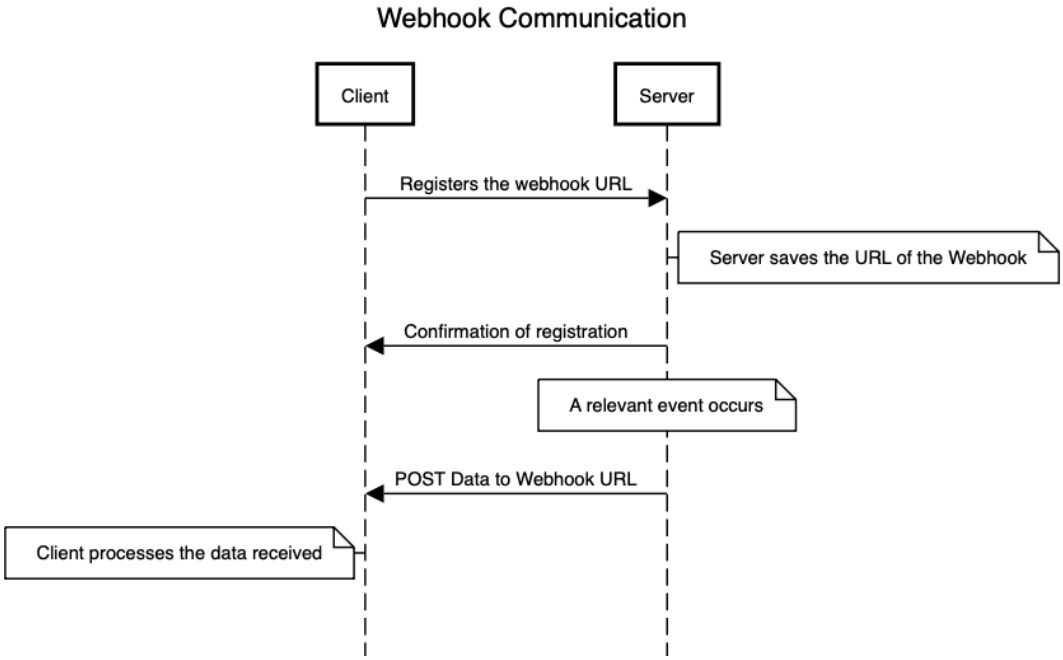



***Figure 6.3:*** *Operation of Webhooks*

This technique is used for payment services, for example, Stripe and Paypal use it to notify third-party applications about payment updates. Chat applications such as Slack use it to integrate notifications from third-party applications and even code versioning applications such as GitHub use Webhooks to notify CI tools of branch pushing operations.

Webhooks are easy to implement because they use the classic request-response paradigm. However, error handling may not be straightforward, for example, if good error handling is not defined for cases where the client is unreachable, it could lead to a loss of updates.

# HTTP Streaming

HTTP Streaming is a technique that enables the continuous flow of data from server to client, allowing users to receive real-time data as it becomes available. This approach is particularly useful for applications that need constant and rapid updates, such as real-time monitoring applications, video or audio streaming, and other forms of dynamic communications.

In general, the server begins transmitting data in response to a client request without closing the connection, sending data to chunks as they become available. An example of an implementation of HTTP Streaming is **HTTP Live Streaming** (**HLS**) used for broadcasting live and on-demand video and audio content.

Another example of an implementation of HTTP Streaming is **Server-Sent Events** (**SSE**), which is a standard designed to be used on any browser that supports HTML5. Clients establish a connection with the server that remains open. The server sends data to the client in the form of messages in UTF-8 format, each having a different identifier, using the MIME type text/event-stream. The diagram in *Figure 6.4* shows how SSE works:

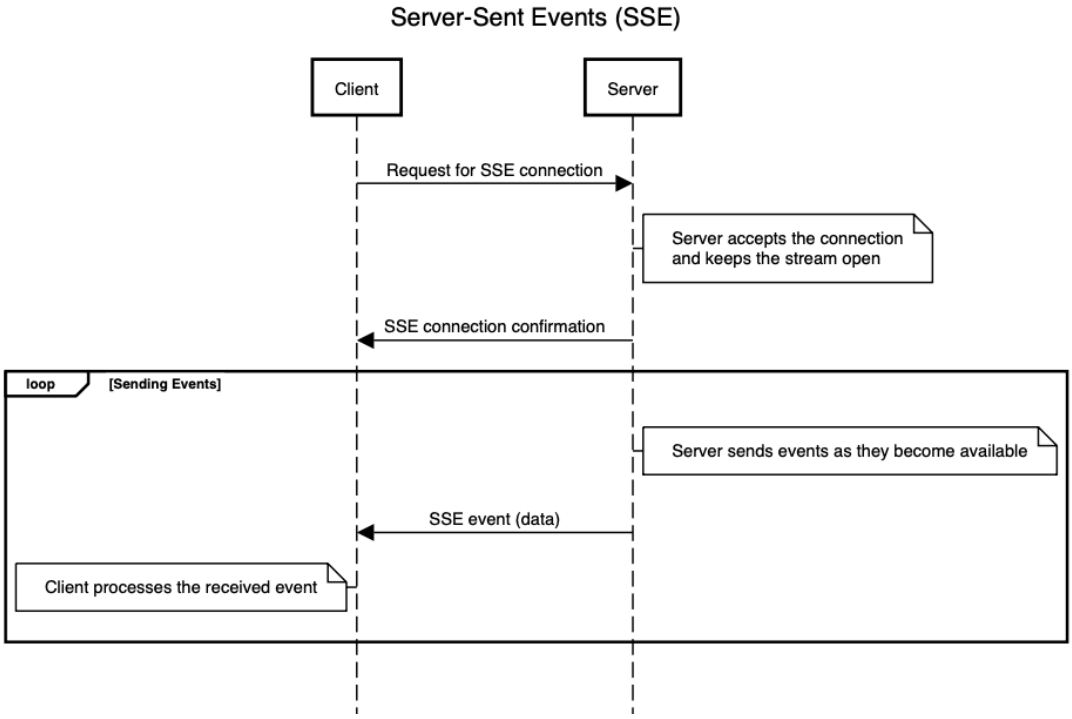



***Figure 6.4**: A sequence diagram showing the operation of SSE*

HTTP Streaming reduces latency in data delivery compared to polling, since data are sent immediately as soon as they are available. It also reduces network overhead on servers since a single connection is used to send messages. Using the HTTP/2 version, it is possible to overcome the unidirectionality of communication (in pure HTTP Streaming, the server sends messages to the client, but not vice versa) and establish communication with greater speed and efficiency. One implementation of HTTP/2 is the gRPC framework. However, it should be kept in mind that not all clients offer support for HTTP/2 and gRPC.

# WebSocket

WebSocket is a full-duplex communication protocol that enables continuous two-way communication between the client and server over a single **Transmission Control Protocol (TCP)** connection. The WebSocket process begins with an HTTP handshake request, during which the client requests to establish a WebSocket connection with the server. If the server accepts the connection, it responds with status code 101. At this point, the connection is updated and passed from HTTP to WebSocket, where the client and server can exchange messages bi-directionally. The diagram in *Figure 6.5* summarizes the operation of WebSocket:

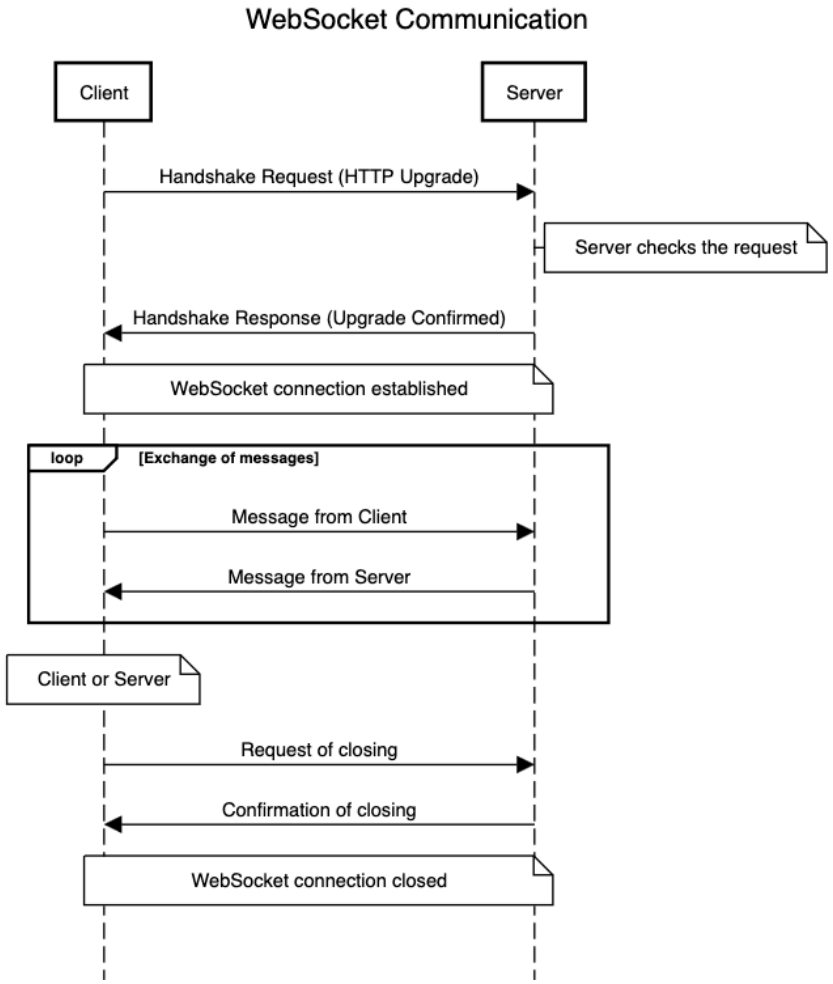



***Figure 6.5**: How WebSocket works*

WebSocket is designed for applications that require high-frequency, low-latency real-time updates, such as real-time chat, multiplayer games, and applications that allow collaboration among multiple users. Compared with HTTP Streaming, it offers lower latency and is more efficient for transmitting large volumes of data. In addition, WebSocket is now compatible with all browsers.

# Publish/Subscribe

The **Publish/Subscribe** (**Pub/Sub**) model is an asynchronous communication design pattern used in software architectures to allow systems and applications to interact in an efficient and scalable manner. In a Pub/Sub system, message producers (publishers) do not send messages directly to specific consumers (subscribers), but rather publish messages to a channel or topic. Subscribers subscribe to these topics and automatically receive relevant messages published on them. The diagram in *Figure 6.6* summarizes how the publish/subscribe pattern works:

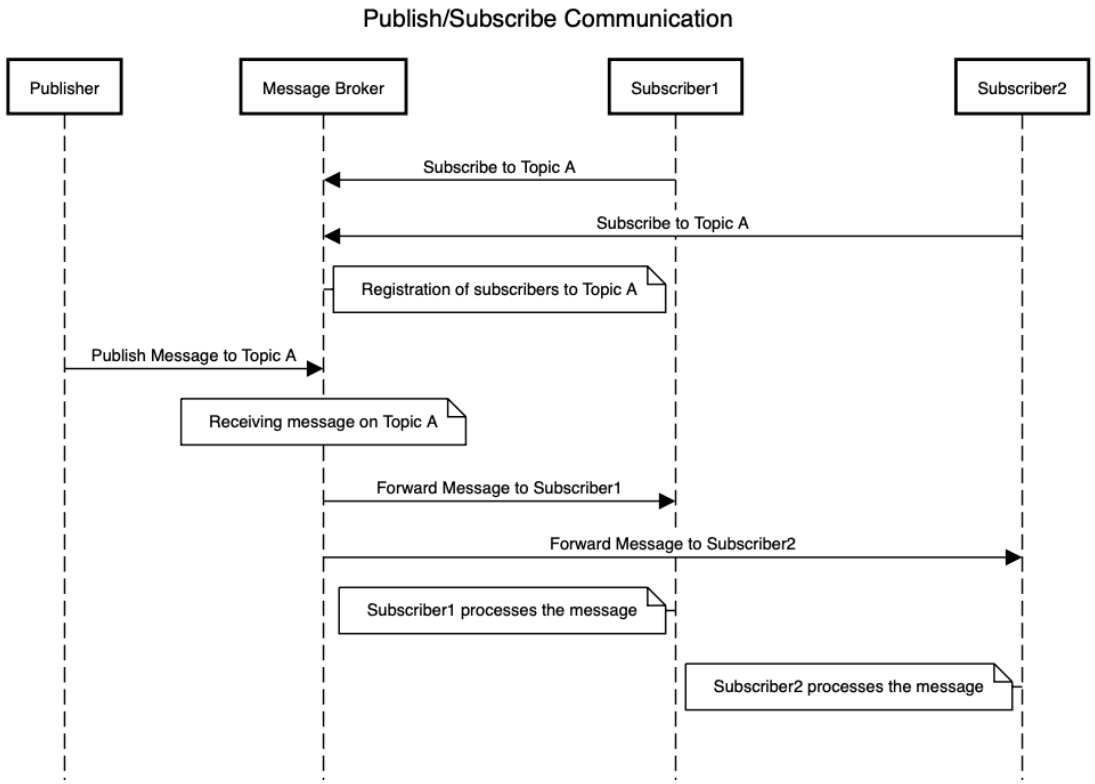



***Figure 6.6**: A diagram showing how the pub/sub pattern works*

Compared with previous approaches, it has the advantage of having a decoupling between the applications that send and those that receive messages: in fact, publishers

and subscribers do not talk to each other directly, but do so through a message broker, allowing greater flexibility and scalability of the system. They send a message on a topic, after which it can be read by the different applications (subscribers) that subscribe to the topic. This decoupling reduces dependencies between components, making it easier to evolve and maintain the system. Another advantage of this approach is that it is easily scalable. The workload on the subscriber can be divided among several instances of the same application. The broker will automatically manage the distribution of messages among the various instances. This approach can be used in a variety of scenarios, such as social media applications that want to notify users of new posts, financial trading systems, and in general communication between microservices where decoupling of components and high horizontal scalability is desired.

# Introduction to Apache Kafka

Apache Kafka is a messaging tool that uses the Pub/Sub pattern. More specifically, it is a distributed event streaming platform that enables publishing, subscribing, archiving, and processing of real-time data streams. The core components of Apache Kafka are:

- **Broker**: It is the middleware that stores events/messages and is concerned with the distribution of those events/messages between producers and consumers. The broker keeps track of the offset of each message that a consumer has read. Offsets represent the logical location of the last message read by the consumer on a specific topic partition, so if a consumer fails, it can resume reading messages at the last point where the failure occurred.
- **Cluster**: It is the set of nodes, i.e., brokers. Having a cluster of nodes allows replication of messages (more precisely, topic partitions) useful in case a node is down. It also allows topic partitions to be distributed among nodes to implement horizontal scaling on brokers.
- **Topic**: It organizes and stores all messages in topics. A topic can be thought of as a folder containing messages. Producers write to topics and consumers subscribe to them to read messages only on the topics to which they have subscribed. The retention time in which a message should remain in the topic is configurable. You can also decide to never delete messages.
- **Partitions:** It divides topics into one or more partitions. When the producer sends an event on a topic, it is stored in a specific topic partition. The default partitioning strategy is one based on the hash of the message key. Otherwise, you may decide to use a round-robin approach or a custom strategy. The messages on a partition are sorted according to the order in which they were inserted.
- **Producer**: It is the client who publishes messages on the topics.
- **Consumer**: It is the client reading the messages from the topics.

- **Consumer group**: It is a group of consumers associated with the same consumer group ID. Consumers in the same consumer group read from different partitions.

In the context of load distribution on message reading, the concepts of partition and consumer group are very important. Indeed, partitions allow simultaneous reading of messages among consumers in the same message group. However, it is important, when creating a topic, to think about the right number of partitions, balancing the advantage that having many can result in better load distribution among consumers, but having too many can result in the disadvantage of using too many resources on brokers. For example, in *Figure 6.7*, topic A has three partitions. The order service application is distributed over three instances, so each instance reads from a different partition. The email service application is distributed over two instances, so one instance reads messages from two partitions, while the third reads from only one partition. In case one wanted to lighten the workload on the email service instances, an additional instance could be added. The shipping service application, on the other hand, has four instances, but since there are only three partitions, the fourth instance will not read from any partition, at least until one shipping service instance goes down. Refer to the following figure for a better understanding:

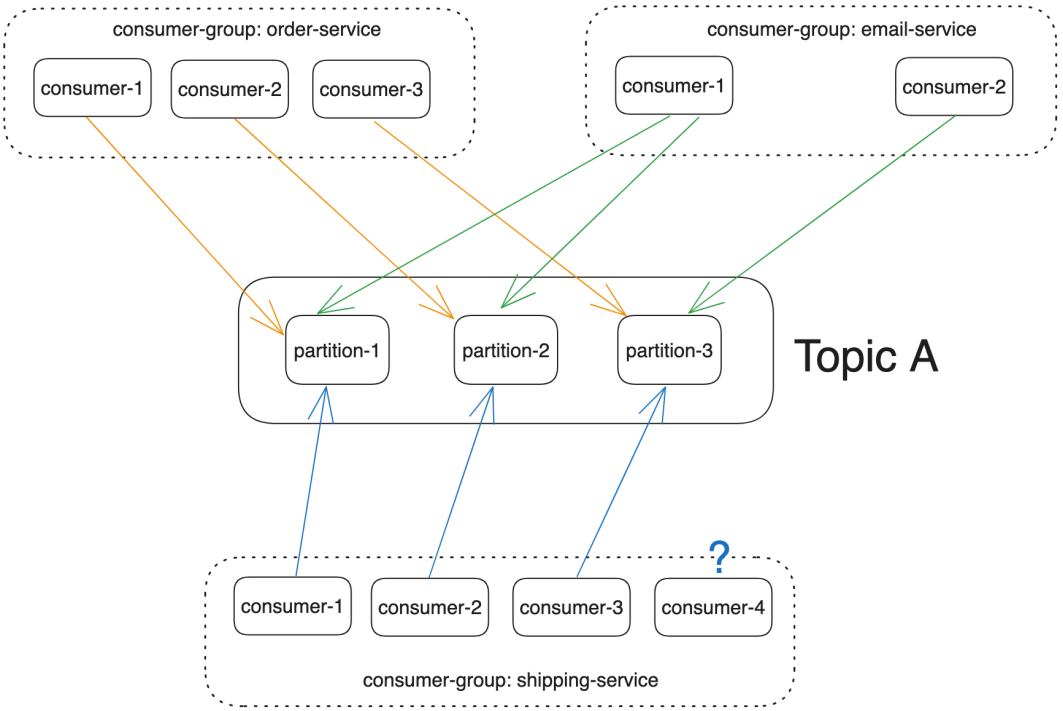



***Figure 6.7**: An example of partition distribution among different consumers*

Now that we have seen the basic concepts of Apache Kafka, we can explore the Spring Cloud Stream project.

# Introduction to Spring for Apache Kafka

This chapter provides a concise overview of Spring for Apache Kafka, a module of the Spring framework designed to simplify the integration of applications with Apache Kafka. By leveraging this module, developers can harness the power of Kafka while adopting the familiar conventions and abstractions of Spring, thereby reducing the complexity of configuration and management. When combined with Spring Boot, the module further minimizes the amount of code required to configure Kafka, automating much of the setup.

To start using Spring for Apache Kafka, you need to include the following Maven dependency in your project:

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
```

The key components of Spring for Apache Kafka are:

- **`KafkaTemplate:`** `It i`s the primary class used to send messages to a Kafka topic. It provides a simple and intuitive interface for producing messages, supporting both synchronous and asynchronous operations.
- **`MessageListenerContainer:`** `It` is an interface responsible for managing the lifecycle of Kafka consumers, including connecting to brokers, polling messages, and dispatching them to methods annotated with **`@KafkaListener.`**
- **`KafkaListener:`** `It` is an annotation that designates a method as a listener for a **`MessageListenerContainer`**. It allows you to configure Kafka consumers by specifying topics, group IDs, and advanced features like error handling**`.`**

Spring Boot greatly simplifies the configuration of a Kafka producer. You only need to add the broker configuration to the **`application.properties`** file, specifying the host and port**`:`**

```
spring.kafka.bootstrap-servers=<broker-1:port>,<broker2:port>
```

If not configured, Spring Boot defaults to localhost:9092. Spring Boot automatically creates a **`ProducerFactory`** bean and a **`KafkaTemplate`** bean, which can be directly used in your application to send messages**`:`**

```
...
@Autowired
private KafkaTemplate<String, String> kafkaTemplate;
```

```
public void sendData(String sensorId, String sensorJson) {
    kafkaTemplate.send("sensor-topic", sensorId, sensorJson);
}
...
```

The send method accepts the topic name, message key, and message body, returning a CompletableFuture to monitor the operation's outcome. Other send methods are available to include additional details, such as partition numbers or timestamps.

For advanced use cases, such as applications requiring multiple Kafka clusters, **`ProducerFactory`** and **`KafkaTemplate`** beans can be manually configured:

```
@Bean
public ProducerFactory<Integer, String> producerFactory() {
    return new DefaultKafkaProducerFactory<>(producerConfigs());
}

@Bean
public Map<String, Object> producerConfigs() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, IntegerSerializer.
class);
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
    // See https://kafka.apache.org/documentation/#producerconfigs for more
properties
    return props;
}

@Bean
public KafkaTemplate<Integer, String> kafkaTemplate() {
    return new KafkaTemplate<Integer, String>(producerFactory());
}
```

To configure a Kafka consumer, specify the broker host and port in the configuration file, as seen earlier, and annotate a method with **`@KafkaListener`** to consume messages. Here is a basic example:

```
@KafkaListener(id = "local-group", topics = "sensor-topic")
public void consumeMessage(String sensorJson) {
    log.info("Consuming message: {}", sensorJson);
}
```

The **`@KafkaListener`** annotation allows you to specify the message group ID with the **`id`** parameter and one or more topics with the **`topics`** parameter**.**

You can also manually configure the **`ConsumerFactory`** bean using the **`DefaultKafkaConsumerFactory`** class:

```
@Bean
public ConsumerFactory<String, String> consumerFactory() {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "local-group");
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
    return new DefaultKafkaConsumerFactory<>(props);
}
```

With Spring for Apache Kafka, you can create Kafka producers and consumers with minimal boilerplate code while leveraging the auto-configuration capabilities of Spring Boot. However, the resulting code remains tightly coupled to the Kafka broker. If, in the future, you decide to switch to another broker like RabbitMQ, part of your application code would need to be rewritten. In the next chapter, we will explore **Spring Cloud Stream**, a project that abstracts the messaging code even further, enabling portability across different messaging systems**.**

# The Spring Cloud Stream project

Spring Cloud Stream is a framework used to create microservices with a message-driven approach. It is an evolution of the Spring Integration framework, which implements Enterprise Integration Patterns (**https://www.enterpriseintegrationpatterns.com**). Spring Cloud Stream allows the developer to focus on business logic rather than using APIs to manage the message broker used. In addition, the latest version of Spring Cloud Stream is based on the Spring Cloud Function project, which promotes the use of functions to implement business logic. Due to the integration with Spring Cloud Function, Spring Cloud Stream has become much easier to use, which is why the use of this framework is becoming increasingly popular.

# An overview of Spring Cloud Function

Spring Cloud Function promotes the use of functions to implement business logic. It allows the execution of functions in both local and PaaS environments such as AWS Lambda. In addition, the implementation of functions is independent of the target runtime used, they can be executed via REST APIs, tasks, and message brokers.

Functions are implemented through the interfaces of **`java.util.function`**, refer to the following points for a better understanding:

- The **`Consumer<T>`** interface allows you to create functions that accept input and do not return output.
- The **`Supplier<T>`** interface allows you to create functions that accept no input parameters but return an output.
- The **`Function<T, R>`** interface allows you to create functions that accept input parameters and return an output.

An example of the use of these interfaces is mentioned below:

```
@Bean
Supplier<String> hello() {
    return () -> "hello";
}

@Bean
Function<String, String> uppercase() {
    return value -> value.toUpperCase();
}

@Bean
Consumer<String> print() {
    return value -> System.out.println(value);
}
```

In addition to the imperative paradigm, Spring Cloud Function also allows functions to be built using the reactive paradigm. Functions are annotated with @Bean since they must be Spring beans. In addition, functions to be used by other frameworks such as Spring Cloud Stream, must be declared via the **`spring.cloud.function.definition`** property, for example:

```
spring.cloud.function.definition=hello;uppercase;print
```

Supplier-type functions used in frameworks such as Spring Cloud Stream represent the source of event streams. They can be either imperative (**`Supplier<T>`**) or reactive (**`Supplier<Flux<T>>`**). Compared to Function and Consumer type functions, taking no input, they are not triggered by an event, partly because they represent the data source themselves. There is a default polling mechanism that executes the imperative functions of type Supplier, every second. Reactive functions, on the other hand, are executed only once, because compared to the imperative version, they produce not a single event but a stream of messages, which could potentially be infinite. However, you may want to poll reactive functions as well, for example, to query data on databases. To do this, just annotate the reactive function with **`@PollableBean`** instead of **`@Bean`**.

Functions of type Function can also be both imperative and reactive. Functions of type Consumer can be both imperative and reactive. Since they are functions that do not return a result, they require further study. If the function is imperative, then the latter returns the void type. Otherwise, if it is reactive, it may return a Flux of events. However, to work, the function must be subscribed, as shown in the following example:

```
@Bean
public Consumer<Flux<String>> print() {
    return fluxEvent -> fluxEvent
            .doOnNext(message -> System.out.println(message))
            .subscribe();
}
```

Alternatively, you can use a function of the following type **`Function<Flux<T>, Mono<Void>>`** to achieve the same result:

```
@Bean
public Function<Flux<String>, Mono<Void>> print() {
    return fluxEvent -> fluxEvent
            .doOnNext(message -> System.out.println(message))
            .then();
}
```

You can also compose several functions into one by using | (pipe) or, (comma) as the function delimiter of the **`spring.cloud.function.definition`** property. Refer to the following example:

```
spring.cloud.function.definition=uppercase|print
```

In the example above, the output of the uppercase function will be the input of the print function.

The composition must follow reasonable logic: you cannot use the output of a function of type Consumer as the input of another function.

Regarding type conversion, Spring Cloud Function uses the **`MessageConverter`** interface to convert incoming data of type **`org.springframework.messaging.Message`** into the type declared in the function and use the **`ConversionService`** interface if the incoming data is not of type Message. However, in most cases, **`MessageConverter`** will always be used since HTTP or messaging requests will be automatically converted to the **`Message`** interface by Spring. The **`Message`** interface consists of the following two methods:

```
public interface Message<T> {
    T getPayload();
    MessageHeaders getHeaders();
}
```

You can choose to use the payload class type directly in your functions, or you can choose to use **`Message`** to get information about the headers as well. However, if not used explicitly, at write time, your payload will be converted to **`Message`**, and at read time, **`Message`** will be converted to your payload, this is due to the **`MessageConverter`** interface, which is as follows:

```
public interface MessageConverter {
    Object fromMessage(Message<?> message, Class<?> targetClass);
    Message<?> toMessage(Object payload, @Nullable MessageHeaders headers);
}
```

# Main concepts in Spring Cloud Stream

Spring Cloud Stream provides an abstraction to the messaging technology being used. The same code can be used to send/receive messages from different message brokers.

The main components of the framework are:

- **Destination binders:** These are the components responsible for providing integration with external messaging systems.
- **Bindings:** It is the bridge between external messaging systems and the application provided by Message Producers and Consumers. Bindings that read data are called **input bindings**, and those that write data are called **output bindings**.
- **Message:** The canonical data structure used by producers and consumers to communicate with Destination Binders.

*Figure 6.8* shows a function that transparently reads from a RabbitMQ queue and writes to an Apache Kafka topic:

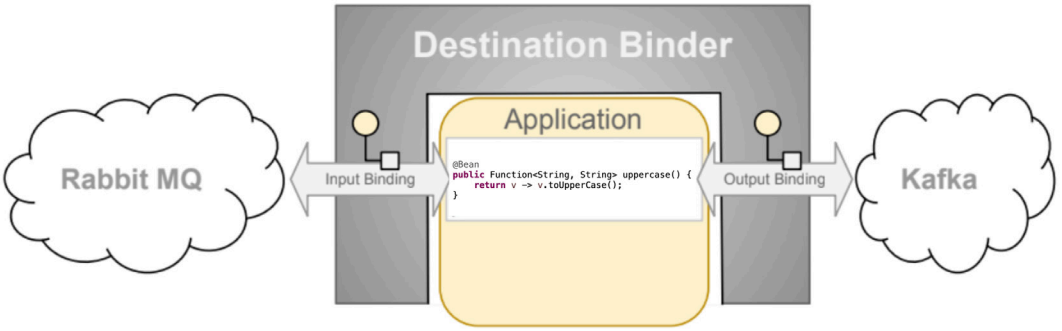



***Figure 6.8**: Diagram of Spring Cloud Stream operation* [1]

The naming of the bindings follows this naming convention:

- **Input bindings: `<functionName>-in-<index>`.**
- **Output bindings: `<functionName>-out-<index>`.**

The index field is the index of the input or output association. It is always zero for functions that take only one input or return only one output. The example below shows a function that takes a string as input and returns the string in uppercase as output:

```
@Bean
public Function<String, String> uppercase() {
    return value -> value.toUpperCase();
}
```

The input bindings will be called **uppercase-in-0** while the output bindings will be called **uppercase-out-0.** To improve readability, you can associate custom names with the bindings using the following property:

**`spring.cloud.stream.function.bindings.<original_name>=<custom_name>.`**

The following shows how to associate a custom name with the Input Binding from the previous example:

**`spring.cloud.stream.function.bindings.uppercase-in-0=input.`**

Once you have associated the custom name, you can use that in all configuration properties.

As for Binders, which are responsible for connecting the application to the message broker, all you have to do is use the appropriate dependency for the message broker you want to use. Spring Cloud Stream provides several modules for different brokers. Others are available from the community. For example, if you wanted to use RabbitMQ, you would need to import the following dependency:

1 **Image source: https://docs.spring.io/spring-cloud-stream/reference/spring-cloud-stream/programming-model.html**

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

Now that we have seen the main concepts of Spring Cloud Stream, we can discuss the framework by putting it into practice.

# Message-driven microservices with Spring Cloud Stream

In this section, we will create a new microservice, a shipping service, responsible for managing the shipping of orders. All the code for this chapter is available in the **chapter-06** folder of the book's GitHub repository.

We can imagine the shipment domain object consisting of the following fields:

- A **tracking code**, which uniquely indicates the shipment of an order.
- An **order code**, which indicates the order to which the shipment is linked.
- A **status**, indicating the status of the shipment. It can mean taking charge, delivering, or being delivered.
- A **shipping date**, indicating when the shipment was created.
- A **delivery date**, indicating when the order was delivered to the customer.

This microservice will communicate with the order service microservice, either synchronously, via REST API, or asynchronously, via messages on a Kafka topic.

In the order creation API, in addition to saving the order entity to the database, the service will also call the **`addShipment`** API of the shipping service. This API will save the shipment entity to the database and simulate some shipment updates of the order by sending two messages on a Kafka topic, **`shipment_event_topic`**, each notifying the shipment status update in delivering and delivered respectively. The two messages are sent with some delay between them. In this way, we will simulate a shipment status update, as if we were using a real external vendor that physically handles the shipment. The order service microservice, will read the message and update the order status.

*Figure 6.9* shows a diagram summarizing the interaction between order service and shipping service:

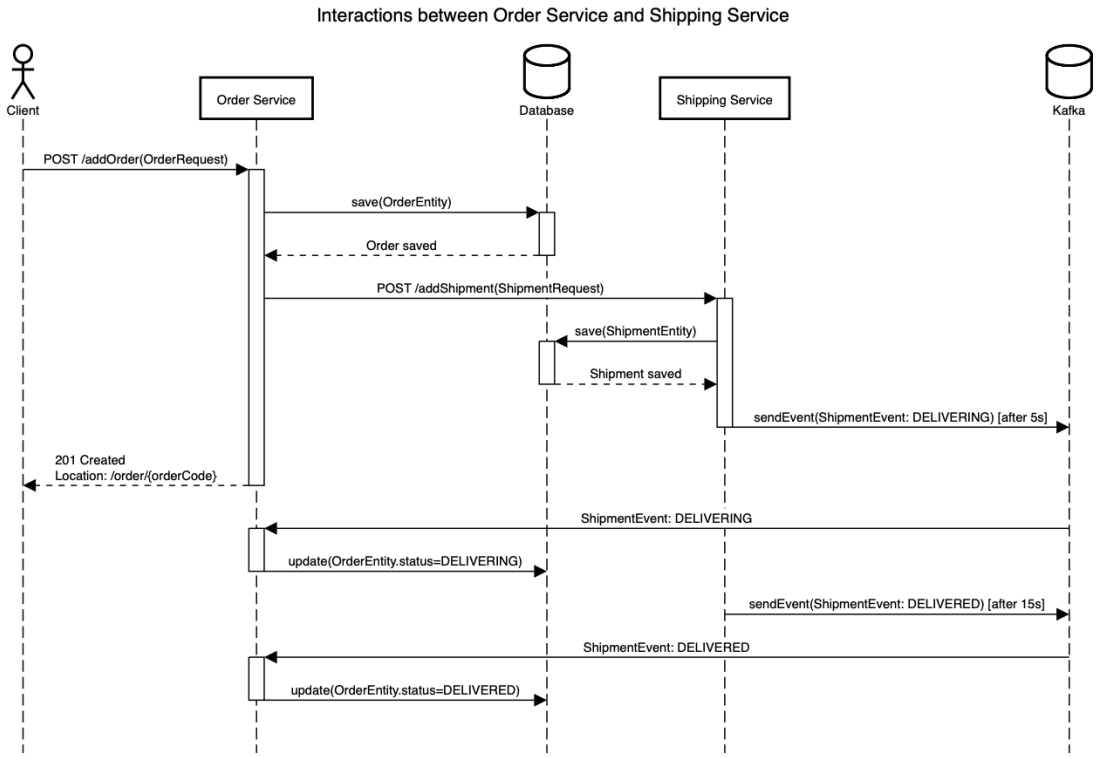



***Figure 6.9**: Interaction between order service and shopping service microservices*

The OpenAPI file of the **`addShipment`** service is as follows:

```
/shipments:
  post:
    tags:
      - shipment
    summary: Add a new shipment
    operationId: addShipment
    requestBody:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/ShipmentRequest'
      required: true
    responses:
      '201':
```

```
        description: Successful operation
        headers:
          Location:
            schema:
              type: string
              format: uri
      '400':
        description: Invalid input
        content:
          application/problem+json:
            schema:
              $ref: '#/components/schemas/ProblemDetail'
...
    ShipmentRequest:
      required:
        - orderCode
      type: object
      properties:
        orderCode:
          type: string
```

The service is very simple: it takes as input, via request body, an order code, and returns a 201 with the URI of the shipment just created.

In the **`infra/docker/init.sql`** file, we add the creation of the table that stores the shipment entity:

```
CREATE DATABASE easyshopdb_shipping;
\c easyshopdb_shipping;
CREATE TABLE IF NOT EXISTS shipments (
      shipment_id SERIAL PRIMARY KEY,
      tracking_code VARCHAR UNIQUE NOT NULL,
      order_code VARCHAR NOT NULL,
      status VARCHAR NOT NULL,
      shipping_date TIMESTAMPTZ,
      delivery_date TIMESTAMPTZ
);
```

Also, because we use Apache Kafka, we modify the following **`docker-compose.yml`** file by adding the Apache Kafka service so that it can be started locally:

```
  broker:
    image: confluentinc/cp-kafka:7.6.1
    hostname: broker
    container_name: broker
    ports:
      - "9092:9092"
      - "9101:9101"
    environment:
      KAFKA_NODE_ID: 1
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: |
        CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS: |
        PLAINTEXT://broker:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_JMX_PORT: 9101
      KAFKA_JMX_HOSTNAME: localhost
      KAFKA_PROCESS_ROLES: 'broker,controller'
      KAFKA_CONTROLLER_QUORUM_VOTERS: '1@broker:29093'
      KAFKA_LISTENERS: |
        PLAINTEXT://broker:29092,CONTROLLER://broker:29093,PLAINTEXT_
HOST://0.0.0.0:9092
      KAFKA_INTER_BROKER_LISTENER_NAME: 'PLAINTEXT'
      KAFKA_CONTROLLER_LISTENER_NAMES: 'CONTROLLER'
      KAFKA_LOG_DIRS: '/tmp/kraft-combined-logs'
      CLUSTER_ID: 'MkU3OEVBNTcwNTJENDM2Qk'
    command: sh -c "((sleep 15 && kafka-topics --bootstrap-
server localhost:9092
      --topic shipment_event_topic --create --partitions 10
      --replication-factor 1)&) && /etc/confluent/docker/run ">
```

The code mentioned above is an adaptation of Confluent's docker-compose: **https://github.com/confluentinc/cp-all-in-one/blob/v7.6.1/cp-all-in-one-kraft/docker-compose.yml**. We added a command that upon creation of the Kafka container automatically creates the **`shipment_event_topic`** topic having ten partitions.

Now that we have taken all the preliminary steps to create the new microservice, all that remains is to write the Java code.

# Getting started with Spring Cloud Stream

This paragraph is divided into two sections, one devoted to the Producer Kafka section, implemented by the new shipping service microservice, and the other devoted to the Consumer Kafka section, implemented by the order service microservice.

## Implementing a producer with Spring Cloud Stream

In order to create the shipping service microservice structure, we use Spring Initializr as usual. We will import the following dependencies as shown in *Figure 6.10*:

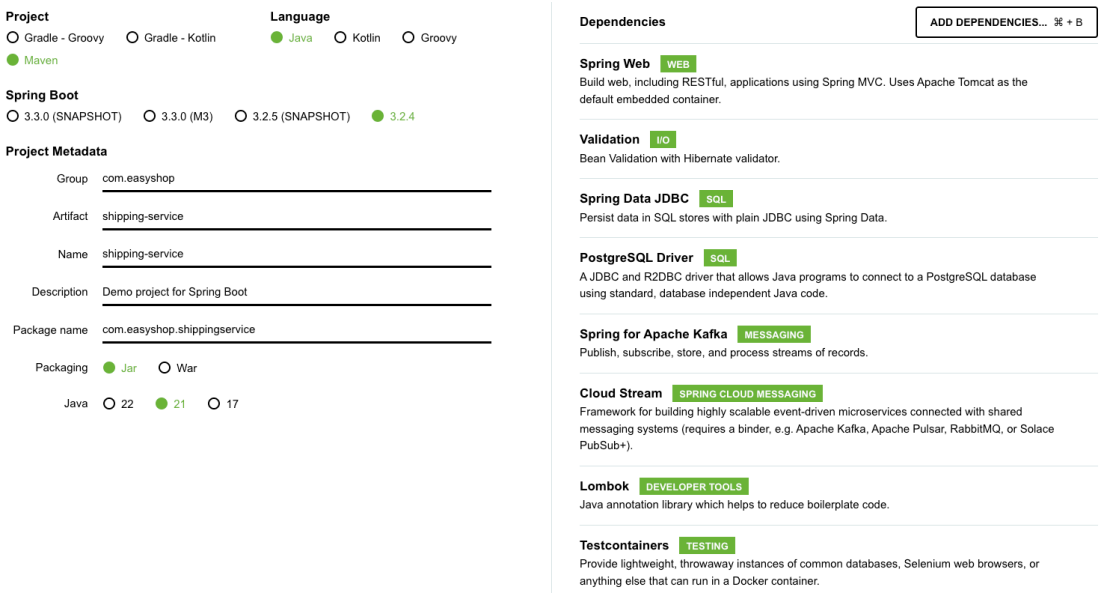



***Figure 6.10**: Spring Initializr screenshot for creation in the new microservice shopping service*

As you can see, compared to the previous chapters, we have added two new dependencies: *Spring for Apache Kafka* and *Cloud Stream*. The first one is for using Apache Kafka with Spring. It would be sufficient to import this dependency if we wanted to use Spring's API dedicated to Kafka (thus without the help of Spring Cloud Stream). The second one is for importing the Spring Cloud Stream project. Having added the Spring for Apache Kafka dependency, Spring Initializr will automatically import the Kafka binder dependency as well, refer to the following code for a better understanding:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-kafka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

Download the project and import the MapStruct, OpenAPI Generator, and Springdoc dependencies, as done for the previous microservices. Also, remember to add the OpenAPI **`shipping-service.yml`** file seen in the previous paragraph.

We start by writing the entity that maps the shipments table and the repository to interact with it, as done for the other microservices. Create the **`middleware.db.entity`** package and write the following Java record:

```
@Table("shipments")
@Builder(toBuilder = true)
public record ShipmentEntity(@Id Long shipmentId,
                             String trackingCode,
                             String orderCode,
                             String status,
                             Instant shippingDate,
                             Instant deliveryDate) {}
```

Create the repo subpackage inside **`middleware.db`** and write the empty repository interface:

```
public interface ShipmentRepository extends CrudRepository<ShipmentEnti-
ty, Long> {

}
```

Now we write the code dedicated to sending the message on the Kafka topic. Let us start with the DTO class that maps the message. Create the **`queue.dto`** subpackage inside the middleware package and write the following record:

```
@Builder(toBuilder = true)
public record ShipmentEvent(
        String trackingCode,
        String orderCode,
        Instant shippingDate,
        Instant deliveryDate,
        String status) {}
```

The order service microservice will read the message and update the order status based on the message status field.

Now, we write the **`ShipmentProducer`** interface, which is responsible for sending the message of type **`ShipmentEvent`**, in the **`middleware.queue.producer`** package:

```
public interface ShipmentProducer {
    boolean publish(ShipmentEvent message);
}
```

The publish signature takes as input the message of type **`ShipmentEvent`** and returns true or false depending on whether the message was sent successfully. The implementation of this interface will use the Spring Cloud Stream API to send the message. Before writing the concrete class, remember that in the previous paragraph, we said that the data source is represented by a function of type Supplier. However, often the data source is not associated with a binder but comes from an external system such as a REST API (which is precisely the case with shipping service). Spring Cloud Stream provides a bean of type **`StreamBridge`** to send data to an output binding. Let us see it at work right away in implementing the **`ShipmentProducer`** interface. Create the impl subpackage within the **`middleware.queue.producer`** package and write the following class:

```
@Component
@RequiredArgsConstructor
@Slf4j
public class ShipmentKafkaProducer implements ShipmentProducer {

    private final StreamBridge streamBridge;

    @Override
    public boolean publish(ShipmentEvent message) {
        log.info("Sending message: {}", message);
        var kafkaMessage = MessageBuilder
                .withPayload(message)
```

```
                .setHeader(KafkaHeaders.KEY, message.orderCode().getBytes())
                .build();
        return streamBridge.send("shipmentEventProducer-
out-0", kafkaMessage);
    }
}
```

The override of the publish method creates an object of type **`Message`**, which encapsulates the payload, of type **`ShipmentEvent`**, and adds the message key in the header, enhancing it with the order code. This means that all messages with the same order code will be sent on the same partition. The send method of the **`StreamBridge`** class takes as input the binding name on which we want to send the message. We will later define the destination (i.e., the Kafka topic name) of the **shipmentEventProducer-out-0** binding in the **`application.properties`** file. However, the send method can also take the topic name as input directly. In that case, a binding will be created at runtime. This use case is useful when you want to send a message on a topic dynamically.

We now create the business logic that saves the shipment entity and sends messages to the order service. Create the service package and write the **`ShipmentService`** class, as shown below:

```
@Service
@RequiredArgsConstructor
@Transactional(readOnly = true)
public class ShipmentService {

    private final ShipmentRepository shipmentRepository;
    private final ShipmentProducer shipmentProducer;
    private final ShipmentMapper shipmentMapper;

    @Transactional
    public String addShipment(ShipmentRequest request) {
        var entity = ShipmentEntity.builder()
                .orderCode(request.getOrderCode())
                .trackingCode(UUID.nameUUIDFromBytes(request
                        .getOrderCode().getBytes()).toString())
                .status(TAKING_CHARGE.getValue())
                .build();
```

```
        var saved = shipmentRepository.save(entity);
        sendShippingEvents(saved);
        return saved.trackingCode();
    }
...
```

The **`addShipment`** method creates an object of type **`ShipmentEntity`** from an order code. After saving it to the database, it invokes the **`sendShipingEvents`** method, which simulates a shipment update. Finally, it returns the generated tracking code to the caller.

The **`sendShippingEvents`** method sends the shipment update messages asynchronously to avoid blocking the main HTTP request thread. This ensures that the tracking code is returned to the client immediately. The implementation of the method is as follows:

```
private void sendShippingEvents(ShipmentEntity entity) {
    var shipmentDate = Instant.now();
    var deliveryDate = shipmentDate.plusSeconds(10);

    var shipStatusFuture = sendEvent(entity, DELIVERING, shipmentDate,
            deliveryDate, 5L);
    var deliveryStatusFuture = sendEvent(entity, DELIVERED, shipmentDate,
            deliveryDate, 15L);

    CompletableFuture.allOf(shipStatusFuture, deliveryStatusFuture);
}

private CompletableFuture<Void> sendEvent(ShipmentEntity entity,
                                          ShipmentStatus status,
                                          Instant shipmentDate,
                                          Instant deliveryDate,
                                          long delaySeconds) {

    return CompletableFuture.runAsync(() -> {

        var updatedEntity = entity.toBuilder()
                .status(status.getValue())
                .shippingDate(shipmentDate)
```

```
                .deliveryDate(deliveryDate)
                .build();

        shipmentRepository.save(updatedEntity);
        var message = shipmentMapper.toMessage(updatedEntity);
        shipmentProducer.publish(message);
    }, CompletableFuture.delayedExecutor(delaySeconds, TimeUnit.SECONDS));
}
```

The method shown above sends a message with status **`DELIVERING`** with a delay of five seconds and then sends a message with status **`DELIVERED`** with a delay of fifteen seconds. Sending the messages is done asynchronously using Java's **`CompletableFuture`** class.

The **`addShipment`** method is called in the controller method of the same name, which implements the OpenAPI Generator's autogenerated interface, ShipmentApi.

The ShipmentMapper mapper is also used, which creates an object of type ShipmentEvent from an object of type ShipmentEntity, as shown below:

```
@Mapper(componentModel = "spring")
public interface ShipmentMapper {
    ShipmentEvent toMessage(ShipmentEntity entity);
}
```

The last step is to add in the **`application.properties`** file the URL of the Kafka broker and the name of the binding target topic:

```
spring.cloud.stream.default-binder=kafka
spring.cloud.stream.kafka.binder.brokers=localhost:9092
spring.cloud.stream.bindings.shipmentEventProducer-out-0.
destination=shipment_event_topic
```

The **`spring.cloud.stream.kafka.binder.brokers`** property in this case is superfluous, since if it is not provided, Spring Cloud Stream by default will try to connect to Kafka right on localhost:9092.

The other properties in the **`application.properties`** file are the usual ones for connecting to the database and using virtual threads, as shown in the code below:

```
spring.application.name=shipping-service
server.port=8083
spring.threads.virtual.enabled=true
spring.datasource.url=jdbc:postgresql://localhost:5432/easyshopdb_shipping
```

```
spring.datasource.username=user
spring.datasource.password=password
```

We have finished the producer implementation; run the command **`./mvnw clean compile`** to autogenerate the OpenAPI Generator's MapStruct code. Let us move on to the consumer implementation.

## Implementing a consumer with Spring Cloud Stream

We add Spring Cloud Stream and Spring for Apache Kafka dependencies to the order service microservice so that it can act as a Kafka consumer:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-kafka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
… and the test dependencies
```

In addition, in the OpenAPI **`order-service.yml`** file, we also add the **`DELIVERING`** status in the **`OrderStatus`** schema:

```
OrderStatus:
  type: string
  enum:
    - TAKEN_CHARGE
    - REJECTED
    - DELIVERING
    - DELIVERED
```

**Note: If our application had already been deployed in production, it would have been fair to create version two of the order service API in order to not break the compatibility with clients.**

Remember to regenerate the code with the command **`./mvnw`** clean compile.

The consumer will read the message sent by the shipping service, so we can copy and paste the **`ShipmentEvent`** record created in the previous subsection, into the **`middleware.queue.consumer.dto`** package.

In the previous paragraph, we mentioned that a consumer in Spring Cloud Stream is implemented by the **`java.function.Consumer`** interface, so we write a class that contains a bean of type **`Consumer<ShipmentEvent>`**. The consumer, once it has read the message, will search the database for the order having the message's order code, check that the shipment status is consistent with the last order status, and finally update the order status on the database. We write the **`ShipmentEventFunctions`** class in the **`middleware.queue.consumer`** package, which reads the message and calls a service method that encapsulates this logic:

```
// ShipmentEventFunctions.java
@Configuration
@Slf4j
@RequiredArgsConstructor
public class ShipmentEventFunctions {

    private final OrderService orderService;

    @Bean
    Consumer<Message<ShipmentEvent>> handleShipmentEvent() {
        return message -> {
            var shipmentEvent = message.getPayload();
            log.info("received shipment event: {}", shipmentEvent);
            orderService.updateOrderByShipment(shipmentEvent);
        };
    }
}

// OrderService.java
...
private final Map<String, String> statusMap = Map.of(
        TAKEN_CHARGE.getValue(), DELIVERING.getValue(),
        DELIVERING.getValue(), DELIVERED.getValue()
);
```

```
...
@Transactional
public void updateOrderByShipment(ShipmentEvent shipmentEvent) {
    orderRepository.findByOrderCode(shipmentEvent.orderCode())
            .map(order -> {
                if (isAConsistentStatus(order, shipmentEvent.status())) {
                    var updatedEntity = order.toBuilder()
                            .status(shipmentEvent.status())
                            .build();
                    return orderRepository.save(updatedEntity);
                }
                return order;
            })
            .orElseThrow(() ->
                    new OrderNotFoundException(shipmentEvent.orderCode()));
}

private boolean isAConsistentStatus(OrderEntity order, String newStatus) {
    return StringUtils.hasText(newStatus) &&
            newStatus.equals(statusMap.get(order.status()));
}
```

As you can see, if the message contains a non-existing order code, an exception is raised. This approach will be useful to us when examining error handling in Spring Cloud Stream. Another aspect to note is that we decided to use the Message interface that wraps the payload of type **`ShipmentEvent`**. This could be useful if we wanted to retrieve header values. However, since we do not perform any logic with them, we could have also used the **`Consumer<ShipmentEvent>`** type as the return value of the method.

Finally, we also need to write the piece of code that implements the REST **`addShipment`** call of shipping service. This is the call that triggers asynchronous communication between the two microservices.

First, in the **`OrderServiceProperties`** class, we add the field indicating the shipping service URL, refer to the following code:

```
@Component
@ConfigurationProperties(prefix = "order.service")
@Data
```

```
public class OrderServiceProperties {

    private String catalogserviceUrl;
    private String shipmentServiceUrl;
    private ProductClientType productClientType;

    public enum ProductClientType {
        REST,
        GRPC
    }
}
```

In order to implement the shipping service REST client, we use the same approach we used for the catalog service one, that is, we will use **`RestClient`** directly. We write the class that acts as the request body, in the package **`middleware.msclient.dto`**, as shown below:

```
@Builder
public record ShipmentRequest(String orderCode) {
}
```

This request body is passed as a parameter in the client interface, which is as follows:

```
public interface ShipmentClient {
    void addShipment(ShipmentRequest request);
}
```

The interface is implemented by the following class that makes HTTP calls using **`RestClient`** directly, refer to the following code:

```
@Component
public class RestClientShipmentClient implements ShipmentClient {

    private final RestClient restClient;

    public RestClientShipmentClient(RestClient.Builder builder,
                                    OrderServiceProperties properties) {
        restClient = builder
                .baseUrl(properties.getShipmentServiceUrl())
                .build();
    }
```

```
    @Override
    public void addShipment(ShipmentRequest request) {
        restClient.post()
                .body(request)
                .retrieve()
                .toBodilessEntity();
    }
}
```

We use the shipping service HTTP client immediately after saving the order, in the **`addOrder`** method of the **`OrderService`** class, which then becomes as follows:

```
...
private final ShipmentClient shipmentClient;

public String addOrder(OrderRequest orderRequest) {
    var totalPrice = calculateTotalPrice(orderRequest);
    var entity = orderMapper.toEntity(orderRequest, totalPrice,
            OrderStatus.TAKEN_CHARGE);
    orderRepository.save(entity);
    shipmentClient.addShipment(ShipmentRequest.builder()
            .orderCode(entity.orderCode())
            .build());
    return entity.orderCode();
}
...
```

All that remains is to add the properties for the correct operation of the consumer. Let us add the following properties in the **`application.properties`** file:

```
order.service.shipment-service-url=http://localhost:8083/shipments

spring.cloud.stream.default-binder=kafka
spring.cloud.function.definition=handleShipmentEvent
spring.cloud.stream.kafka.binder.brokers=localhost:9092
spring.cloud.stream.kafka.binder.consumer-properties.max.poll.records=250
spring.cloud.stream.bindings.handleShipmentEvent-in-0.destination=shipment_
event_topic
```

```
spring.cloud.stream.bindings.handleShipmentEvent-in-0.group=${spring.
application.name}
```

The first Spring Cloud Stream property is used to specify what the default binder is, in case the project imports multiple binders. The second is used to specify the functions to be used in the framework. The third allows us to define the URL (or URLs, if we have a cluster of nodes) of the Kafka broker. The fourth allows us to specify the maximum number of records returned by a single **`poll()`** call, that is, the call that the Kafka API makes to retrieve records from a topic. In general, we can use the following nomenclature to set the native Kafka consumer properties: **`spring.cloud.stream.kafka.binder.consumer-properties.<kafka_property>`**. You can take a look at this link to learn about all the properties available for Kafka consumers: **https://docs.confluent.io/platform/current/installation/configuration/consumer-configs.html**. Similarly, you can set properties for kafka producers using **`spring.cloud.stream.kafka.binder.producer-properties.<kafka_property>`**. The last two properties are used to specify the name of the topic on which the consumer with the binding name handleShipmentEvent-in-0 should read and the name of the consumer's **`groupId`**. All consumers with the same **`groupId`** will read from different partitions (they are therefore a part of the same message group).

We have written all the necessary code for managing the shipment of orders, all that remains is to start the microservices.

# Spring Cloud Stream at work

We start the Postgres and Kafka containers using the command **`docker compose up -d`** from the path **`infra/docker`**. After that, we start the microservices of catalog service, order service, and shipping service. To verify that the asynchronous communication between the order service and shipping service works properly, we need to create a new order. We then make the following HTTP call to create a new order:

```
http :8081/orders \
  Content-Type:application/json \
  "products[0][productCode]"=0001 \
  "products[0][quantity]":=2 \
  "products[1][productCode]"=0002 \
  "products[1][quantity]":=1
```

Let us analyze the shipping service logs:

```
Sending message: ShipmentEvent[trackingCode=75afe79c-8256-3cbe-
8014-617700a31c72, orderCode=AJPIV1717237573473, shippingDate=2024-
06-01T10:26:13.745578Z, deliveryDate=2024-06-
01T10:26:23.745578Z, status=DELIVERING]

...
```

```
Sending message: ShipmentEvent[trackingCode=75afe79c-8256-3cbe-
8014-617700a31c72, orderCode=AJPIV1717237573473, shippingDate=2024-
06-01T10:26:13.745578Z, deliveryDate=2024-06-
01T10:26:23.745578Z, status=DELIVERED]
```

We can see that two messages are correctly sent for the same order a few seconds apart, one with **`DELIVERING`** status and one with **`DELIVERED`** status.

Let us look at the order service logs:

```
received shipment event: ShipmentEvent
[trackingCode=75afe79c-8256-3cbe-8014-
617700a31c72, orderCode=AJPIV1717237573473, status=DELIVERING, shippingDat-
e=2024-06-01T10:26:13.745578Z, deliveryDate=2024-06-01T10:26:23.745578Z]


received shipment event: ShipmentEvent
[trackingCode=75afe79c-8256-3cbe-8014-
617700a31c72, orderCode=AJPIV1717237573473, status=DELIVERED, shippingDate-
=2024-06-01T10:26:13.745578Z, deliveryDate=2024-06-01T10:26:23.745578Z]
```

We can see that the order service correctly received the messages, from the logs. Try creating a new order. Then, from the command line run the following command:

```
docker exec -t broker /usr/bin/kafka-console-consumer --bootstrap-
server localhost:9092 --topic shipment_event_topic --from-
beginning --property print.key=true --property print.partition=true
```

This command allows reading the contents of a Kafka topic, and printing, in addition to the message payloads, the keys, and partitions. The command should return output like this:

```
Partition:1     UPN7K1717239324694      {"trackingCode":"49e05ddb-
c9a2-3d9e-9d21-
32d44e12ba5f","orderCode":"UPN7K1717239324694","shippingDate":"2024-0-
6-01T10:55:24.929870Z","deliveryDate":"2024-06-01T10:55:34.929870Z","status
":"DELIVERING"}
Partition:1     UPN7K1717239324694      {"trackingCode":"49e05ddb-
c9a2-3d9e-9d21-
32d44e12ba5f","orderCode":"UPN7K1717239324694","shippingDate":"2024-0-
6-01T10:55:24.929870Z","deliveryDate":"2024-06-01T10:55:34.929870Z","status
":"DELIVERED"}


Partition:8     AJPIV1717237573473      {"trackingCode":"75afe79c-
8256-3cbe-8014-
617700a31c72","orderCode":"AJPIV1717237573473","shippingDate":"2024-0-
```

```
6-01T10:26:13.745578Z","deliveryDate":"2024-06-01T10:26:23.745578Z","status
":"DELIVERING"}
Partition:8     AJPIV1717237573473      {"trackingCode":"75afe79c-
8256-3cbe-8014-
617700a31c72","orderCode":"AJPIV1717237573473","shippingDate":"2024-0-
6-01T10:26:13.745578Z","deliveryDate":"2024-06-01T10:26:23.745578Z","status
":"DELIVERED"}
```

The output shows that the messages in the same order are part of the same partition. You could have done the same verification by printing, in the consumer code, the header with the key **`kafka_receivedPartitionId`**.

You have successfully completed the implementation of asynchronous communication between two microservices using Spring Cloud Stream.

# The AsyncAPI specification

Before we move on to the error handling section, although we will not go into it in this book, we would like to introduce you to an open-source initiative called **AsyncAPI**. The latter is a specification for documenting asynchronous APIs (i.e., message-driven APIs), taking a cue from the OpenAPI initiative. The specification is agnostic to the protocol used, so it can be used for WebSockets, Kafka, AMQP, MQTT, etc. Let us look at the main components of the specification in practice, using the order service consumer API.

Similar to OpenAPI, the AsyncAPI document begins with version information of the specification used and descriptive information:

```
asyncapi: 3.0.0
info:
  title: Shipment Event Consumer
  version: 1.0.0
  description: |
    This service consumes shipping events from Apache Kafka and up-
dates the status of orders in the database.
```

The **`server`** section is used to specify information about the message broker used, such as the URL and protocol, as shown in the following code:

```
servers:
  kafkaServer:
    host: localhost:9092
    description: Kafka server running locally
    protocol: kafka
```

The **operations** section is used to specify the name and type of the operation (send/receive), as well as the channel (i.e., for Kafka, the topic on which to receive the message). The optional bindings part, inside operations, is used to indicate information about the operation specific to the message broker used, such as the name of the **groupId**:

```
operations:
  handleShipmentEvent:
    action: receive
    channel:
      $ref: '#/channels/shipmentEventTopic'
    bindings:
      kafka:
        bindingVersion: '0.5.0'
        groupId:
          type: string
          enum: [ 'operation-service' ]
```

The **channels** section is used to specify the channel on which the application writes or reads messages. Again, the bindings portion is used to indicate information specific to the Kafka channel, such as the number of topic partitions. Refer to the following code for a better understanding:

```
channels:
  shipmentEventTopic:
    description: Kafka channel for shipping events
    address: shipment_event_topic
    messages:
      shipmentEvent:
        $ref: '#/components/messages/ShipmentEventMessage'
    bindings:
      kafka:
        bindingVersion: '0.5.0'
        partitions: 10
        replicas: 1
```

The components section, similar to OpenAPI, is used to define the structure of the objects utilized by operations:

```
components:
  messages:
```

```
  ShipmentEventMessage:
    bindings:
      kafka:
        key:
          type: string
          description: 'The orderCode from the payload'
    payload:
      $ref: '#/components/schemas/ShipmentEvent'
    headers:
      properties:
        id:
          type: string
          description: 'The orderCode from the payload'

schemas:
  ShipmentEvent:
    type: object
    properties:
      trackingCode:
        type: string
        description: Shipment tracking code
      orderCode:
        type: string
      status:
        type: string
      shippingDate:
        type: string
        format: date-time
      deliveryDate:
        type: string
        format: date-time
    required:
      - trackingCode
      - orderCode
      - status
```

You can find the complete file in the **asyncapi** folder. You can learn more about the AsyncAPI specification at the following link: **https://www.asyncapi.com/docs**.

# Error handling in Spring Cloud Stream

If an exception is raised in the consumer, it is propagated to the binder, which makes several retries of the same message (by default the total number of retries is 3, including the first one). Retries are handled automatically using the *Spring Retry* library, imported from Spring Cloud Stream. You can change the number of retries on a single binding using the property:

**spring.cloud.stream.bindings.<binding_name>.consumer.max-attempts.**

If this property is set to 1 and the DLQ is not enabled, then the default container retries the number of Spring for Apache Kafka, which is 10. However, if all retries fail, two handlers are used by default, the first one, logs the message, while the second one depends on the type of binder used. In this case, the second handler simply deletes the message.

Let us look at error handling in Spring Cloud Stream in practice. We implemented the order service consumer in such a way that it raises an exception if it receives a message with a non-existing order code. To simulate this use case, we will manually send a message on the Kafka topic.

In order to send a message on the Kafka topic, we run the following command from the command line:

```
docker exec -it broker /usr/bin/kafka-console-producer \
--bootstrap-server localhost:9092 --topic shipment_event_topic \
--property parse.key=true --property key.separator=:
```

The command allows you to send a message by specifying the message key and payload, separated by: (semicolon). We will enter the following record:

```
not-exist:{"orderCode": "not-exist", "status": "DELIVERING"}
```

We analyze the order service logs to verify that it handled the error as we expected:

```
2024-06-01T19:03:15.980+02:00
received shipment event: GenericMessage [payload=...}]
2024-06-01T19:03:16.993+02:00
received shipment event: GenericMessage [payload=...}]
2024-06-01T19:03:19.007+02:00
received shipment event: GenericMessage [payload=...}]

LoggingHandler   : org.springframework.messaging.
MessageHandlingException...Caused by: com.easyshop.orderservice.exception.
```

```
OrderNotFoundException: Order with code not-exist not found

DefaultErrorHandler   : Backoff none exhausted for shipment_
event_topic-3@8...Caused by: com.easyshop.orderservice.exception.
OrderNotFoundException: Order with code not-exist not found
```

Effectively, the message is processed three times. Since all three attempts failed, the error is then handled by the logging handlers and the default Kafka handler. One aspect to note is the time interval between attempts. Since the default backoff interval is set to 1000 milliseconds, the second attempt is made after one second from the first. Also, since the default backoff multiplier is set to 2.0, the third attempt is made about two seconds after the previous one (1000 milliseconds x 2.0).

You can change the value of these parameters by setting the property **`spring.cloud.stream.bindings.<binding_name>.consumer.back-off-initial-interval`** and **`spring.cloud.stream.bindings.<binding_name>.consumer.back-off-multiplier`**.

There are situations when it is not enough to log the discarded message. Spring Cloud Stream makes it easy to handle discarded messages by sending them in a **Dead Letter Queue** (DLQ). To do this, simply set the property **`spring.cloud.stream.kafka.bindings.<binding_name>.consumer.enable-dlq`** to true. Optionally, you can supply the DLQ topic name with the **`spring.cloud.stream.kafka.bindings.<binding_name>.consumer.dlq-name`** property. If this property is not valued, Spring Cloud Stream will try to send the discarded message on a topic having the name **`error.<original_topic_name>.<gruop_id>`**. We then add in the **`application.properties`** file, the configurations for the DLQ:

```
spring.cloud.stream.kafka.bindings.handleShipmentEvent-in-0.consumer.
enable-dlq=true

spring.cloud.stream.kafka.bindings.handleShipmentEvent-in-0.consumer.dlq-
name=shipment_event_topic_dlq
```

Resend the previous message in the topic and look at the order service logs. Kafka's default handler has been replaced by DLQ handling. You can verify that the DLQ is working properly by reading the message in the **`shipment_event_topic_dlq`** topic by running the following command:

```
docker exec -t broker /usr/bin/kafka-console-consumer \
--bootstrap-server localhost:9092 --topic shipment_event_topic_dlq \
--from-beginning --property print.key=true --property print.partition=true
```

You should see the following discarded message:

```
Partition:3    not-exist    {"orderCode": "not-exist", "status": "DELIVERING"}
```

Spring Cloud Stream also allows you to implement custom error handlers by defining beans of type **`Consumer<ErrorMessage>`**.

The **`org.springframework.messaging.support.ErrorMessagge`** class contains the exception that generated the error and the original message. Let us see in practice how a custom error handler for order service works. In the **`ShipmentFunctions`** class, we write a method that handles consumer **`handleShipmentEvent`** errors:

```
@Bean
Consumer<ErrorMessage> handleShipmentEventError() {
    return message -> {
        Throwable error = message.getPayload();
        Message<?> originalMessage = message.getOriginalMessage();
        MessageHeaders headers = originalMessage.getHeaders();
        var originalMessageKey = (byte[]) headers.get(KafkaHeaders.
RECEIVED_KEY);
        var originalMessagePayload = (byte[]) originalMessage.getPayload();
        log.
info("Original MessageKey: {}, MessagePayload: {}, with error: {}",
                new String(originalMessageKey, StandardCharsets.UTF_8),
                new String(originalMessagePayload, StandardCharsets.UTF_8),
                error.getCause().getMessage());
        //Save to db or do something else....
    };
}
```

In this custom error, we simply log the error, the message key, and its payload, but we could easily implement more complex handling, such as saving the message key and payload on some error table. Remember that headers also have other useful information besides the message key, such as the original topic name. To apply the custom error handler, we comment out the two properties added earlier for DLQ handling and add the following property:

```
spring.cloud.stream.bindings.handleShipmentEvent-in-0.error-handler-definiti
on=handleShipmentEventError
```

If you want to apply this custom error handler to all consumers instead of one, you can do so by using this property:

```
spring.cloud.stream.bindings.upper.error-handler-definition=
handleShipmentEventError
```

**Note: If you did not comment on the two properties to handle the error in DLQ, the custom error handler would still replace the DLQ behavior, but it would be confusing to have both handlers set up.**

Resend the previous message; you should see in order service such a log, certifying the operation in our custom error handler, refer to the following code:

```
Original MessageKey: not-exist, MessagePayload: {"orderCode": "not-
exist", "status": "DELIVERING"}, with error: Order with code not-
exist not found
```

In this section, you have seen how with Spring Cloud Stream, it allows you to handle errors in a really simple way, giving you a choice of several strategies. It is up to you to decide which way is most appropriate based on the nature of your application.

**Note: The error handling seen in this paragraph applies only to Message handlers, that is, imperative functions. If you use the reactive approach, you need to use the Reactive APIs to handle errors (for example, retryWhen to handle retries).**

# Testing Spring Cloud Stream applications

Spring Cloud Stream provides a test binder that can be used to test framework components without the need to use an actual binder. The test binder can be used by importing the following dependency:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-test-binder</artifactId>
    <scope>test</scope>
</dependency>
```

**Note: When this book was written, Spring Initializr imported version 2023.0.1 of the parent module spring-cloud-dependencies. However, we recommend upgrading the version to a later one, for example, 2023.0.2. In version 2023.0.1 the test binder is used by default for tests, and this can generate problems if you want to run tests with a real binder (see https://github.com/spring-cloud/spring-cloud-stream/issues/2931).**

We test the order service consumer by exploiting the test binder. Create the **`middleware.queue.consumer`** test package and write the following class:

```
@SpringBootTest
@EnableAutoConfiguration(exclude = {DataSourceAutoConfiguration.class})
@ActiveProfiles("test")
@Import(TestChannelBinderConfiguration.class)
```

```
class ShipmentEventFunctionsTest {

    @Autowired
    private InputDestination input;

    @Autowired
    private ObjectMapper objectMapper;

    @MockBean
    private OrderRepository orderRepository;

    @MockBean
    private Consumer<ErrorMessage> handleShipmentEventError;

}
```

As we annotate the class with **`@SpringBoot`**, the test class initializes all Spring context. However, we do not need to import the datasource context, so we exclude datasource configurations with **`@EnableAutoConfiguration`**. To use the test binder, we import the configuration of the **`TestChannelBinderConfiguration`** class. We also enable the test profile with **`@ActiveProfiles("test")`** such that JDBC auditing is disabled. We then modify the following code of **`DataConfig`** class in this manner:

```
@Configuration
@EnableJdbcAuditing
@Profile("!test")
public class DataConfig {
}
```

We now turn to writing a test method. The **`TestChannelBinderConfiguration`** provides the **`InputDestination`** and **`OutputDestination`** classes for testing input and output bindings, respectively. In this case, since the order service is a consumer, we will use **`InputDestination`**. In the test class, we write a method that verifies the correct operation of the consumer upon receiving a message with an existing order code:

```
@Test
void testReceivedMessageOk() throws IOException {
    var event = ShipmentEvent.builder()
            .orderCode("001")
            .status(OrderStatus.DELIVERED.getValue())
            .build();
```

```
    var message = MessageBuilder
            .withPayload(objectMapper.writeValueAsBytes(event))
            .setHeader(KafkaHeaders.KEY, "001")
            .build();

    var entityBefore = OrderEntity.builder().orderCode("001")
            .status(OrderStatus.DELIVERING.getValue()).build();
    var entityAfter = OrderEntity.builder().orderCode("001")
            .status(OrderStatus.DELIVERED.getValue()).build();

    when(orderRepository.findByOrderCode("001"))
            .thenReturn(Optional.of(entityBefore));
    when(orderRepository.save(entityAfter))
            .thenReturn(entityAfter);

    this.input.send(message, "shipment_event_topic");

    verify(orderRepository).findByOrderCode("001");
    verify(orderRepository).save(entityAfter);
}
```

The test builds a message and sends it using the send method of the object of type **`InputDestination`**. This method takes the message as input to be sent and the destination, which is the topic name. Try running the test; it should work correctly. To test the case of receiving a message with an order code, not in the database, we can verify that the consumer reads the message three times (since the max-attempts are set to three). There is an additional check that we can make, if we have enabled the custom error handler, it is to verify that it has also been invoked. The test method will then be similar to the following code:

```
...
@MockBean
private Consumer<ErrorMessage> handleShipmentEventError;
...

@Test
void testReceivedMessageKo() throws IOException {
```

```
    var event = ShipmentEvent.builder()
            .orderCode("not-exists")
            .status(OrderStatus.DELIVERED.getValue())
            .build();

    var message = MessageBuilder
            .withPayload(objectMapper.writeValueAsBytes(event))
            .setHeader(KafkaHeaders.KEY, "not-exists")
            .build();

    when(orderRepository.findByOrderCode("not-exists"))
            .thenReturn(Optional.empty());

    this.input.send(message, "shipment_event_topic");

    verify(orderRepository, times(3)).findByOrderCode("not-exists");
    verify(orderRepository, never()).save(any());
    verify(handleShipmentEventError).accept(any(ErrorMessage.class));
}
```

Another way to test our consumers is to use the true binder. In all projects where we have imported the **Testcontainers** dependency, this is a great time to use it. In the test folder, we write a class that centralizes the Postgres and Kafka configurations:

```
@TestConfiguration
public class IntegrationEnvConfig {

    static KafkaContainer kafka = new KafkaContainer(DockerImageName
            .parse("confluentinc/cp-kafka:7.6.1"))
            .withKraft();

    static PostgreSQLContainer<?> pg = new PostgreSQLContainer<>(DockerImag
eName
            .parse("postgres:16.2"))
            .withDatabaseName("easyshopdb_order")
            .withUsername("user")
            .withPassword("password");

    static {
        pg.start();
```

```
        kafka.start();
        System.setProperty("spring.kafka.bootstrap-servers",
                kafka.getBootstrapServers());
        System.setProperty("spring.cloud.stream.kafka.binder.brokers",
                kafka.getBootstrapServers());
        System.setProperty("spring.datasource.url", pg.getJdbcUrl());
    }
}
```

The **`KafkaContainer`** and **`PostgreSQLContainer`** classes allow us to specify the container image name and configurations. These classes are usable as our project has the following dependencies:

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>kafka</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <scope>test</scope>
</dependency>
```

The static block starts the two containers and overrides the value of the Spring properties from Postgres and Kafka using the dynamic values from Testcontainers. Now, you can import this class into any class that runs integration tests by annotating the class with **`@Import(IntegrationEnvConfig.class)`**. Containers will be shared among the different test classes that import this configuration. We leave it up to you to implement the integration test and the shipping service test using the **`OutputDestinataion`** class (you can still find them in the GitHub repository for this chapter).

# Conclusion

In this chapter, we implemented asynchronous communication between two microservices using the Spring Cloud Stream framework and the Apache Kafka message broker. With Spring Cloud Stream, you can implement producer and consumer as simple Java functions, since it is based on the Spring Cloud Function framework. The great advantage of this framework is that it hides all the complexity of managing your chosen message broker. Also, the same source code can be used regardless of the message broker chosen. Although

in this chapter we used the Apache Kafka binder, you can easily switch to another binder such as RabbitMQ. The framework also allows you to manage multiple binders in the same project. You could create a function that reads from one broker and writes to another broker. Error handling in Spring Cloud Stream is also very simple. Use the Spring Retry module to perform retries before going into the error channel. You can choose to use the default error handler, a custom error handler, or put the message in DLQ. Like every Spring framework, Spring Cloud Stream also provides a dedicated testing module that makes it easy to test input and output bindings via the InputDestination and OutputDestination classes. By reading this chapter you now have all the knowledge to implement message-driven microservices with Spring Cloud Stream. At this point in the book, we have learned how to implement synchronous and asynchronous APIs with Spring. However, a key aspect of designing an application is securing the API. In the next chapter, we will see how to do this using Spring Security with OpenID Connect and OAuth2.

# Points to remember

- Spring Cloud Stream allows business logic to be written agnostic to the chosen message broker, using Spring Cloud Function.
- Spring Cloud Stream, in case of an error, by default attempts to process the message three times before going to the error channel.
- You can use Spring Cloud Stream with either the imperative or reactive paradigm. However, if you choose to use the reactive paradigm, error handling must be implemented using the Reactive API.

# Exercises

1. Implement shipping service tests using the OutputDestination class.
2. Implement the integration test using the Kafka binder and the Testcontainers library.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

**https://discord.bpbonline.com**



# Chapter 7

# Centralized Security with Spring Cloud Gateway

## Introduction

In this chapter, we will look at a critical aspect of API management, namely securing APIs. Although this topic is placed after API implementation, in the real world it is important to think about security aspects already during the design phase and not after the implementation phase. Spring provides the Spring Security module to implement API security. This module supports several security modes. In this chapter, we will show you how to implement API security using the **OpenID Connect (OIDC)** 1 and OAuth 2.1 specifications. Although there is the Spring Authorization Server module to implement an **identity server** (**IS**), in this chapter we will use Keycloak, an open-source IS developed by Red Hat. However, you will see that using one IS over another is completely transparent to Spring Security.

In the first section of the chapter, we will see an overview of the concepts of OIDC and OAuth2, we will look at the types of OAuth2 flows. For our project, we will choose the authorization code flow.

Next, we will create a new microservice, edge service, using the Spring Cloud Gateway, Spring's module for implementing an API Gateway. The API Gateway plays a key role in hiding downstream services from the outside world, acting as a reverse-proxy, but not only that, it also handles cross-cutting concerns. We will see how to make the Easyshop application resilient through Resilience4J with Spring Cloud Gateway, applying retry and

circuit breaker patterns. In addition, the API Gateway will act as an OAuth2 client, while the catalog service microservice will act as an OAuth2 resource server.

Readers can refer to the GitHub repository in the **`chapter-07`** folder.

# Structure

In this chapter, we will discuss the following topics:

- Introduction to OAuth2 and OpenID Connect
- Using Keycloak as an identity server
- API Gateway with Spring Cloud Gateway
- OAuth2 client and resource servers with Spring Security

# Objectives

In this chapter, you will learn how to implement and manage security in a microservices architecture using Spring Cloud Gateway and OAuth2. You will explore the core concepts of OAuth2 and OIDC, including their roles in enabling secure authentication and authorization across distributed systems. The chapter covers the setup and integration of Keycloak, a popular open-source identity and access management solution, to centralize user authentication and manage clients, realms, and roles.

You will discover how to use Spring Cloud Gateway as an API Gateway to secure and manage traffic between microservices, configure routes, filters, and predicates, and enforce security policies. Additionally, you will gain hands-on experience in setting up microservices as OAuth2 clients and resource servers using Spring Security, implementing OIDC authentication, managing access tokens, and securing endpoints with fine-grained access control.

By the end of this chapter, you will have the knowledge and practical skills to centralize security in your microservices architecture, ensuring robust API protection and a seamless user experience.

# Introduction to OAuth2 and OpenID Connect

To secure APIs, two concepts must be applied such as authentication and authorization. Authentication answers the question of **who wants to access the API**, and authorization answers the question of **does the user (or, in general, the client), has permission to invoke the API.** Obviously, authorization cannot occur if the client has not authenticated first. In this chapter, we will create the edge microservice that will act as the API Gateway, shielding the downstream services as a catalog service and using standards such as OIDC, OAuth2, and **JSON Web Token (JWT)** to secure the API. *Figure 7.1* shows the figure summarizing the parts involved in this chapter:

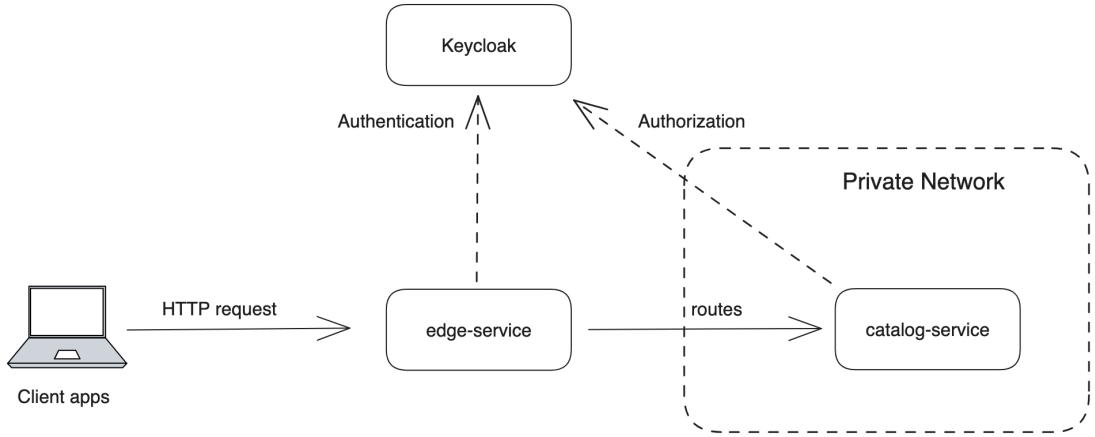



***Figure 7.1:*** *The flow we will implement in this chapter using an API Gateway and an IS*

# JSON Web Token

Before exploring OIDC and OAuth2, it is important to understand what JWT is. JWT is an open standard (RFC 7519) for securely representing claims between two parties. It is a signed token that contains information about the authenticated user. The token consists of three parts encoded in Base64 and separated by the dot (.) character. These parts are as follows:

```
<header>.<payload>.<signature>
```

The **`header`** is a JSON object that usually contains information about the token type, the algorithm used for signing, and a key identifier:

```
{
  "alg": "HS256",
  "typ": "JWT",
  "kid": "6Gk-Ym…"
}
```

The **payload** contains **claims**, which are information about the logged-in user and additional information. Some claims are mandatory, others are recommended. Custom claims can also be defined. An example of a payload is as follows:

```
{
  "iss": "https://exaple.issuer.com",
  "sub": "mrossi",
  "exp": 1719056069,
  "roles": ["admin"]
}
```

The first field indicates the **issuer**, that is, who generated the token. The **`sub`** field indicates the subject, that is, an identifier of the logged-in user. The **`exp`** field indicates the expiration of the token. The tokens must be short-lived for security reasons. Finally, the **`roles`** field is a custom claim that defines the roles of the logged-in user.

The JWT **signature** is used to verify that the payload has not been altered. To create the signature, the encoded header, the encoded payload, and a secret are taken, after which, using the algorithm specified in the header, the signature is applied. For example, if it were decided to use the **HMAC SHA256** algorithm, the signature would be created in the following way:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

JWTs can be signed using a secret or using a public and private key pair.

The prerequisite of those consuming a JWT is that they trust the issuer, that is, the entity that generated the token.

Actually, what we have seen is a JWT of type **JSON Web Signature (JWS).** If the claims are confidential and, therefore, need to be encrypted by the authorization server and decrypted by the resource server, a token of type **JSON Web Encryption (JWE)** can be used. However, for our application, we will use JWS.

# OpenID Connect protocol

OIDC is an identity layer on top of the **OAuth2** protocol that allows an application, called **Client**, to verify the identity of the end user, who identifies himself on an authorization server. When authentication is successful, the authorization server returns end-user information to the client application in the form of a JWT called an **Identity Token (ID Token)**. The authentication flow of OIDC is based on the **authorization code flow** of OAuth2.

The main actors in the OIDC stream are the following:

- **User**: User is a person who uses a registered client application to access resources. In OIDC, it is also called an **end-user**. In OAuth2, it is not necessarily a person. It is called **Resource Owner, OAuth2 User**, or even **end-user** as in OIDC.
- **Client**: Client is an application that requires from an authorization server both an ID token to authenticate the user and an access token to allow the user to access resources. To do this, the client needs to be registered on the authorization server. In OIDC, the client is also called a **Relying Party (RP).** In OAuth2, it is called the **OAuth2 client**.
- **Authorization server**: It is an entity that implements the OIDC and OAuth2 protocols. It is also called an **Identity Provider (IP),**

The flow of OIDC is based on the **authorization code flow** of OAuth2, and that, in the Easyhop application, is realized in the following steps:

1. When an unauthenticated user tries to access an API exposed by edge service, edge service redirects the browser to the Keycloak login page.
2. The user enters his credentials, and if valid, Keycloak redirects the browser to the edge service callback **URL**, which contains, among the query parameters, the **authorization code**.
3. Edge service invokes the Keycloak token endpoint to exchange the received authorization code with an **ID token** and an **access token**.
4. Upon receiving a positive response from the token endpoint, the edge service initializes an authenticated user session with the browser using a session cookie and maintains a mapping between the session and the ID token.

The flow of OIDC applied to our system on a user not yet authenticated is shown in *Figure 7.2*:

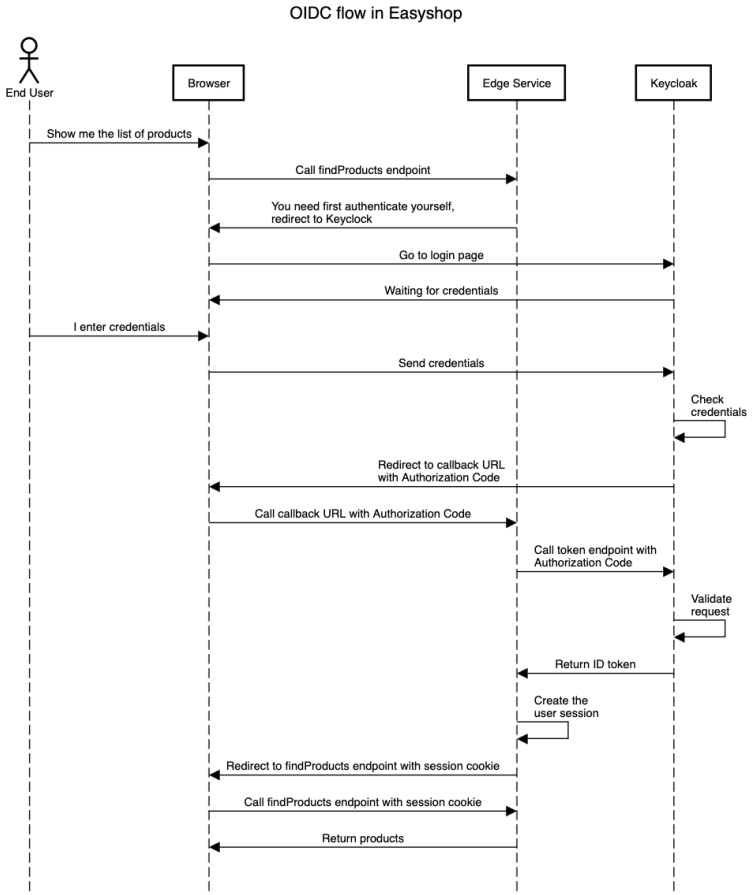



***Figure 7.2:***

# OAuth2 protocol

OAuth2 is an industry-standard protocol for authorization that is designed to allow easy integration for any type of application-web, desktop, mobile, or machine-to-machine. OAuth2 allows a user to grant a third-party application limited access to protected resources without sharing credentials.

OAuth2 defines four actors (or roles). Three have already been discussed in the *OIDC section*. The fourth is the **resource server**, the server that hosts protected resources. It accepts the access token in requests to understand whether the user is allowed to access the resource. Compared to the ID token, it is not mandatory that the access token be a JWT, although we will use JWT for our application. However, the access token must be a short-lived token. The authorization server, optionally, in addition to the access token, can also return a **refresh token**, a token with a longer duration, which allows you to request a new access token, when the one used has expired.

In the Easyshop application, the OAuth2 client is an edge service, while the resource server is a catalog service. However, there are cases where an application acts as both an OAuth2 client and resource server. For example, in Spotify you can log in with your Google account. In that case, Google acts as the IP, while Spotify acts as both the OAuth2 client and resource server. *Figure 7.3* illustrates the application of OAuth2 in Easyshop:

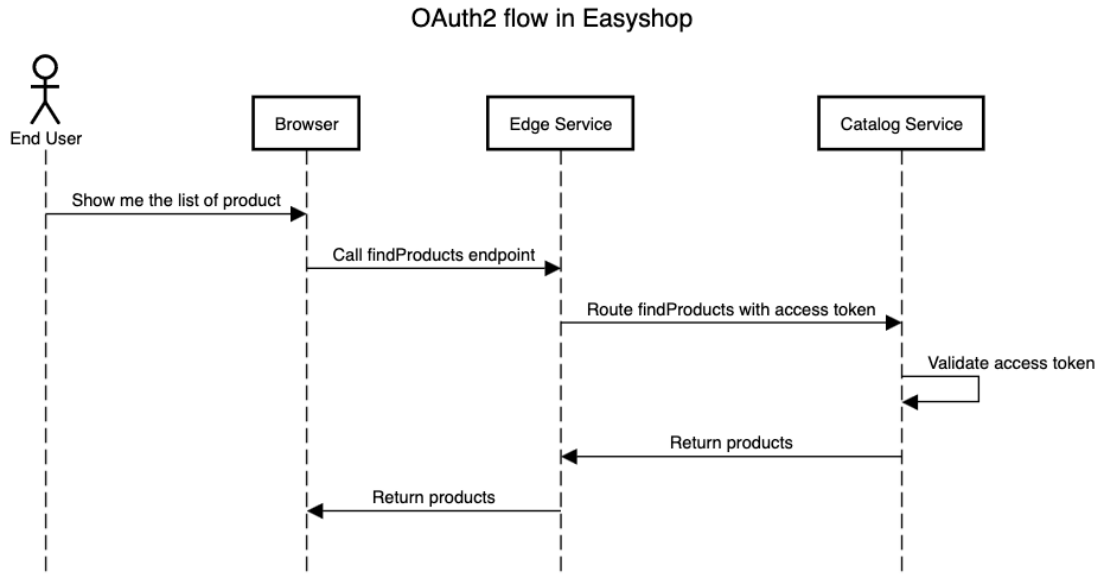



***Figure 7.3:*** *The OAuth2 flow from the perspective of the resource server*

From the figure above, you might wonder how the catalog service can validate the access token without calling the authorization server. The first time the catalog service needs to validate the access token, it actually calls the **JSON Web Key Set (JWKS)** endpoint of the

```
{
  "keys": [
    {
      "kty": "RSA",
      "kid": "1234abcd",
      "use": "sig",
      "n": "0vx7agoebGcQSuuPiLJXZptN27...T9yEoQ",
      "e": "AQAB"
    }
  ]
}
```

Catalog service uses the public key of the list having the kid field that matches the kid of the access token to verify the validity of the access token. The result of the call to the JWKS endpoint is cached to save calls to the authorization server for subsequent requests. If the IP rotates keys, the resource server, when it receives the access token, will not find the kid value among the cached ones, so it will call the JKWS endpoint again and update the cache.

However, the OAuth2 flow seen so far is the authentication code flow. However, the following are other types of flows that we will not read in this chapter:

- **Implicit**: A simplification of the authorization code flow optimized for clients implemented in a browser such as JavaScript clients. In the implicit flow, the IP returns the ID token immediately, skipping the step of exchanging between an authorization code and the ID token.
- **Resource owner password credentials:** Allows an access token to be derived using credentials directly. Use only when there is a high degree of trust between the resource owner and the client application.
- **Client credentials:** Allows an access token to be obtained using clientId and client secret. Useful when the client is also the resource owner.

Now that we have seen the theoretical concepts of security, we are ready to put them into practice, with Keycloak and Spring Security.

**Note: Choosing which component should be the OAuth2 client is not easy, and there is no single right choice. For example, we could have chosen to use the frontend application that consumes the edge service API as the OAuth2 client. However, choosing a frontend application such as OAuth2 client would imply giving access token management to the latter, which is not secure. My advice is as follows: if the frontend is a SPA, then it is better to choose a backend application such as OAuth2 client. If the frontend is a mobile app, it might be a good idea to choose the latter as an OAuth2 client.**

# Using Keycloak as an identity server

In this chapter, we will see how to start Keycloak locally and how to configure it to censor users and roles. The role will be useful to us in the authorization phase, since, in addition to using OAuth2, we will also use **Role-Based Access Control (RBAC)**, which is useful for assigning permissions to users based on the role they have. Specifically, we will implement a mechanism whereby users with the **manager** role can use both the read and write APIs of catalog service, while users with the **user** role can invoke only the read API.

# Running Keycloak locally

Keycloak is an open-source Identity and Access Management (IAM) solution developed by Red Hat that supports various protocols such as OIDC, OAuth2, and Security Assertion Markup Language (SAML).

The advantage of using an IAM tool is that it relieves our application of imported security concepts, such as how to register and census users and manage passwords. In addition, by using OIDC, we can give the user the option to authenticate via a registered account or via social login. In both cases, our application is not impacted.

The easiest way to run Keycloak is through containers. We then add the **`docker-compose.yml`** file and the Keycloak service:

```
keycloak:
  image: quay.io/keycloak/keycloak:25.0
  container_name: "easyshop-keycloak"
  command: start-dev --import-realm
  volumes:
    - ./keycloak:/opt/keycloak/data/import
  environment:
    - KC_DB=postgres
    - KC_DB_URL=jdbc:postgresql://easyshop-postgres:5432/keycloak
    - KC_DB_USERNAME=user
    - KC_DB_PASSWORD=password
    - KEYCLOAK_ADMIN=user
    - KEYCLOAK_ADMIN_PASSWORD=password
  ports:
    - 8091:8080
  depends_on:
    - pg
```

**Note: A volume is used since we have exported the Keycloak configurations that we will make in this paragraph via script. You can avoid having to configure Keycloak every time you initialize the database. The volume file is located in the `infra/docker/Keycloak` folder.**

By default, Keycloak uses the H2 database. However, since we already use Postgres for our application, we configure the IS to use it.

Add in the **`infra/docker/postgres`** folder the file **`postgres_keycloak.sql`** that contains the instructions to create the database dedicated to the IS:

```
DROP DATABASE IF EXISTS keycloak;

CREATE DATABASE keycloak;
```

To ensure the entire folder is imported instead of a single file, update the volume section of the **`pg`** service as follows:

```
volumes:
  - ./postgres:/docker-entrypoint-initdb.d
  - easyshop-postgres:/var/lib/postgresql/data
```

To start Keycloak, run the command **`docker compose up -d`** from the path **`infra/docker`**.

# Configuring Keycloak

The Keycloak components we will use are the following:

- **Realms**: They are identity management spaces. Each realm manages users, credentials, and roles. A user registered in one realm cannot authenticate in another realm.
- **Clients**: They are the applications that interact with Keycloak. For example, in our case, the client application will be edge service.
- **Users**: These represent the entities that can authenticate themselves. Users can be managed internally by Keycloak or come from external IPs.
- **Roles**: They are used to define permissions. They can be assigned directly to users or groups.

We will configure Keycloak via the admin **Command Line Interface** (**CLI**), although the same settings can be configured via **Graphical User Interface** (**GUI**) by accessing the console from the browser at **`http://localhost:8091`**.

1. First, we enter the Keycloak container filesystem using the following command:

   ```
   docker exec -it easyshop-keycloak bash
   ```

2. Let us place ourselves in the **`/opt/keycloak/bin`** folder using the following command:

   ```
   cd /opt/keycloak/bin/
   ```

3. Log in as admin using the following command:

```
./kcadm.sh config credentials --server http://localhost:8080 \
--realm master --user user --password password
```

4. After that, we create the realm for Easyshop:

```
./kcadm.sh create realms -s realm=Easyshop -s enabled=true
```

5. On this realm, we will now create roles and users. To create the manager and user roles, we perform the following instructions:

```
./kcadm.sh create roles -r Easyshop -s name=manager
./kcadm.sh create roles -r Easyshop -s name=user
```

6. We create two users, one will have the role of manager, and the other user is as follows:

```
./kcadm.sh create users -r Easyshop -s username=john \
-s email=john@mail.
com -s firstName=John -s lastName=Smith -s enabled=true

./kcadm.sh create users -r Easyshop -s username=mrossi \
-s email=mrossi@mail.
com -s firstName=Mario -s lastName=Rossi -s enabled=true
```

7. We set the following two users a password:

```
./kcadm.sh set-password -r Easyshop --username john \
--new-password john

./kcadm.sh set-password -r Easyshop --username mrossi \
--new-password mrossi
```

8. Finally, to the user john we assign the role of manager, and to the user mrossi, we assign the role of the user:

```
./kcadm.sh add-roles -r Easyshop --uusername=john --rolename manager
./kcadm.sh add-roles -r Easyshop --uusername mrossi --rolename user
```

We will configure the client application later when we create the edge service project.

# API Gateway with Spring Cloud Gateway

An API Gateway is a crucial component in the architecture of modern systems, especially in the context of microservices and distributed architectures. It functions as a unified entry point for all API calls in a system. This tool provides several essential functionalities called

**cross-cutting concerns**, such as authentication management, resilience management, monitoring data collection, and performance analysis, in addition to the simple operation of routing requests to downstream services.

Spring provides the Spring Cloud Gateway project to implement an API Gateway. There is an imperative version, which uses Spring MVC, and a reactive version, which uses WebFlux. We will use the one with Spring WebFlux.

The core elements of Spring Cloud Gateway are the following:

- **Route**: The basic gateway element, consisting of an ID, a destination URI, a set of predicates, and filters. A route is matched if the aggregate predicate is true.
- **Predicate**: Predicates based on the HTTP request, such as headers or parameters.
- **Filter**: Filters are that allow requests and responses to be modified before or after the request is sent to downstream services.

Routes can be configured in Spring Cloud Gateway using either the Java Routes API or configuration files. Below are examples of both approaches:

```
// Using the Java Routes API
@Configuration
class RouteConfiguration {

    @Bean
    public RouterFunction<ServerResponse> gatewayRouterFunctionsCookie() {
        return route("cookie_route")
                .route(header("X-Request-Id", "\\d+"),
                        http("https://example.org"))
                .build();
    }
}

# Using configuration file
spring:
  cloud:
    gateway:
      mvc:
        routes:
        - id: header_route
          uri: https://example.org
```

```
        predicates:
        - Header=X-Request-Id, \d+
```

For the complete list of predicates, you can take a look at the official documentation **https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway/request-predicates-factories.html**.

Now that we have made a theoretical introduction to Spring Cloud Gateway, all that remains is to create the edge-service microservice.

**Note: We chose to use the version of Spring Cloud Gateway with WebFlux because, at the writing of this chapter, the implementation with Spring MVC is less stable and has fewer features than the reactive version. For example, the MVC version lacks default filters: https://github.com/spring-cloud/spring-cloud-gateway/issues/3177, or there are problems with circuit breaker configuration: https://github.com/spring-cloud/spring-cloud-gateway/issues/3327.**

# Getting started with Spring Cloud Gateway

As with the other microservices, we create the skeleton of the edge service project using Spring Initializr. The dependencies to import are Reactive Gateway, which allows you to import the reactive version of Spring Cloud Gateway, Resilience4J, which allows you to apply patterns such as circuit breaker to improve the resilience of the application, and Lombok, which allows us to write cleaner code. You can import the project from GUI or by running the following command:

```
curl https://start.spring.io/starter.zip -d groupId=com.easyshop \
  -d artifactId=edge-service -d name=edge-service \
  -d packageName=com.easyshop.edgeservice \
  -d dependencies=cloud-gateway-reactive,cloud-resilience4j,lombok \
  -d javaVersion=21 -d bootVersion=3.2.4 -d type=maven-project \
  -o edge-service.zip
```

Note: Throughout this chapter, we will import more dependencies into edge-service. In the **`README.md`** file of the project, you will find the command above that already includes all the necessary dependencies.

Although so far we have used the **`application.properties`** file to set up configurations, Spring also allows the YML format file to be used for the same purpose. For edge service, we will use the **`application.yml`** file. So delete the **`application.properties`** file inside the resources folder and create the **`application.yml`** file with the following content:

```
spring:
  application:
```

```
    name: edge-service
server:
  port: 8090
```

The piece of code above would correspond, in the **`application.properties`** file, to these assignments:

```
spring.application.name=edge-service
server.port=8090
```

We create the route that allows requests to be routed to the catalog service microservice using the following codes:

```
spring:
  application:
    name: edge-service
  cloud:
    gateway:
      routes:
        - id: catalog-service
          uri: http://localhost:8080/products
          predicates:
            - Path=/products/**
```

The route with catalog-service id linked to the endpoint **http://localhost:8080/products** is invoked if the path of the request URL to edge service starts with **`/products`**. We run the edge service and catalog service and try to invoke the **`findProductByCode`** API of the catalog service via the edge service:

```
http :8090/products/LAP-ABCD
```

We will get the following response:

```
HTTP/1.1 200
{
    "brand": "FirstBrand",
    "category": "laptop",
    "code": "LAP-ABCD",
    "name": "ALaptop",
    "price": 30000
}
```

The basic configuration worked perfectly. However, what happens if the catalog service microservice is down? We should make our API Gateway resilient to the tastes of downstream services like catalog service.

# Resilience with Spring Cloud Gateway

To make the Easyshop system resilient to failures, we can implement retry and circuit breaker patterns at the entry point of our application, such as edge service.

## Implementing retry with retry filter

The retry pattern is useful to obviate situations where the target service is temporarily unavailable.

To implement it, we will use the **Retry Filter** provided by Spring Cloud Gateway. Filters can be associated with specific routes, or they can be associated with all routes, in this case, they are called **default-filters**. We will use the second approach for the Retry filter. The Retry filter supports the following parameters:

- **retries:** Indicates the number of retries after the first failure.
- **methods:** Indicates which HTTP methods to make retries for.
- **series:** Indicates the series of status codes for which to make retries. The series are represented by the enum **`org.springframework.http.HttpStatus.Series`**.
- **exceptions:** A list of exceptions for which to make retries.

In addition, you can specify the **backoff** parameters that we have already seen in previous chapters.

An example of a retry filter configuration that we could apply is as follows:

```
spring:
  application:
    name: edge-service
  cloud:
    gateway:
      default-filters:
        - name: Retry
          args:
            retries: 3
            methods: GET
            series: SERVER_ERROR
```

```
        exceptions: |
    java.io.IOException,java.net.ConnectException,
    org.springframework.cloud.gateway.support.TimeoutException
        backoff:
          firstBackoff: 100ms
          maxBackoff: 1s
          factor: 2
          basedOnPreviousValue: false
routes:
  ...
```

If the parameter **`basedOnPreviousValue`** is set to **`false`**, the backoff formula is **`firstBackoff * (factor ^n)`** otherwise, the formula is **`prevBackoff * factor`**. By default, the parameter is set to **`true`**.

We have seen how to easily implement the retry pattern in Spring Cloud Gateway to overcome situations of momentary unavailability of the target service. However, to handle situations where the target service is unavailable for an extended time, the circuit breaker pattern must be applied.

## Implementing circuit breaker with Resilience4J

The circuit breaker pattern is an architectural design pattern used to improve the resilience and stability of a distributed system. This pattern is inspired by electrical circuits, where a switch (circuit breaker) is used to interrupt the flow of power in the event of an overload or failure. Similarly, in a software context, a circuit breaker interrupts the flow of requests to a service that is experiencing problems, preventing the fault from propagating and damaging the entire system.

The circuit breaker pattern consists of the following three states:

- **Closed**: In this state, all requests to the downstream service are allowed. The circuit breaker monitors the success and failure of requests. If the number of failures exceeds a predefined threshold, the circuit breaker switches to the **Open** state.
- **Open**: In this state, all requests to the downstream service are blocked to avoid overloading the already struggling service. After a configurable timeout period, the circuit breaker switches to the **Half-Open** state.
- **Half-Open**: In this state, the circuit breaker allows a limited number of requests toward the downstream service to check if the problem is resolved. If the number of requests that have failed is equal to or greater than a configured threshold, the circuit breaker returns to the **Open** state, otherwise, it returns to the **Closed** state.

An activity diagram summarizing the behavior of the circuit breaker pattern is shown in *Figure 7.4:*

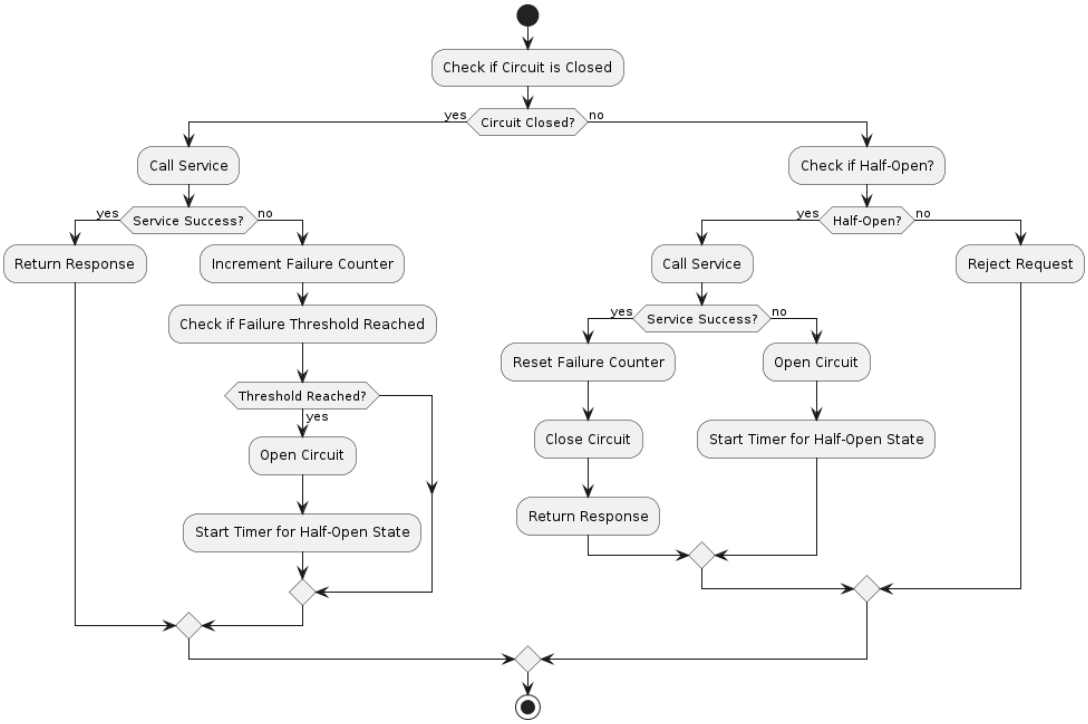



***Figure 7.4:*** *The activity diagram showing state management in the Circuit Braker pattern*

To implement the circuit breaker, we use the route created in the edge service, with the CircuitBreaker filter, which uses the Spring Cloud CircuitBreaker API. These APIs support several libraries. However, the Resilience4J library is supported out of the box. By adding the Resilience4j dependency in Spring Initializr, we will find ourselves with the following imported module:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>
        spring-cloud-starter-circuitbreaker-reactor-resilience4j
    </artifactId>
</dependency>
```

The automatically imported module is the reactive version since we used the reactive version of Spring Cloud Gateway.

The CircuitBreaker filter takes as arguments the name and a possible **`fallbackUri`**, a

circuit breaker parameters can be configured using Resilience4j properties. One possible configuration is as follows:

```
      routes:
        - id: catalog-service
          uri: http://localhost:8080/products
          predicates:
            - Path=/products/**
          filters:
            - name: CircuitBreaker
              args:
                name: catalogServiceCircuitBreaker
                fallbackUri: forward:/catalog-service-fallback


resilience4j:
  circuitbreaker:
    configs:
      default:
        slidingWindowSize: 10
        permittedNumberOfCallsInHalfOpenState: 2
        failureRateThreshold: 50
        waitDurationInOpenState: 15000
```

Let us analyze the following parameters of Resilience4J:

- **`slidingWindowSize`**: This parameter defines the size of the sliding window used to calculate the failure rate. In this, the circuit breaker evaluates the state of the last ten calls made.
- **`permittedNumberOfCallsInHalfOpenState`**: This parameter specifies the number of calls that are allowed during the **Half-Open** state. When the circuit breaker switches from the **Open** to **Half-Open** state, it allows two calls to check whether the service has returned to proper operation. If the two calls are successful, the circuit breaker returns to the **Closed** state, otherwise if at least one fails, it returns to the **Open** state.
- **`failureRateThreshold`**: This parameter defines the failure rate threshold above which the circuit breaker switches to the **Open** state. In this case, if more than 50% of the calls in the scroll window fail, the circuit breaker will open.

- **`waitDurationInOpenState`**: This parameter specifies the duration (in milliseconds) for which the circuit breaker remains in the **Open** state before switching to the **Half-Open** state to attempt some test calls.

These parameters have been placed in the default configurations **`(resilience4j.circuitbreaker.configs.default)`**, so they will apply to all circuit breakers. The configurations can be overridden for a given circuit breaker using the **`resilience4j.circuitbreaker.configs.<circuitbreaker_name> property.`**

Regarding the **`fallbackUri`** parameter, all calls made during the Open state of the circuit will be redirected to the **`/catalog-service-fallback`** URI. We can create in edge-service a **`RestController`** to handle these calls as follows:

```
//package com.easyshop.edgeservice.api
@RestController
@RequestMapping(("catalog-service-fallback"))
public class FallbackController {

    @GetMapping
    public Mono<ResponseEntity<Object>> fallbackCatalogServiceGet() {
        return Mono.just(ResponseEntity.noContent().build());
    }

    @PostMapping
    @PutMapping
    @DeleteMapping
    public Mono<ResponseEntity<Object>> fallbackCatalogServiceNotGet() {
        return Mono.just(ResponseEntity
                .status(HttpStatus.SERVICE_UNAVAILABLE)
                .build());
    }
}
```

In this example, calls made by the GET method on catalog service during the open circuit will return a 204 and, for other methods, a 503. In a real system, you might want to implement a more complex mechanism, such as managing a cache in such a way that GET calls, during the open loop, return the last value where the call was successful.

To test the operation of the circuit breaker, you can set the Resilience4J logs to DEBUG in the **`application.yml`** file as follows:

```
logging:
  level:
    io.github.resilience4j: DEBUG
```

We invoke the catalog service API via edge service with the catalog service microservice turned off. The following command demonstrates how to make ten requests using Apache Benchmark:

```
ab -n 10 http://localhost:8090/products/LAP-ABC
```

To observe the behavior of the circuit breaker, we examine the edge service logs. After the tenth call fails, the logs indicate a change in circuit state from Closed to Open, as shown below:

```
Event STATE_TRANSITION published: 2024-07-01T18:40:14.798500+02:00[Eu-
rope/Rome]: CircuitBreaker 'catalogServiceCircuitBreak-
er' changed state from CLOSED to OPEN
```

To test the system after the catalog service has been restarted, start the catalog service and then make two requests to the edge service using the following command:

```
ab -n 2 http://localhost:8090/products/LAP-ABC
```

To observe the circuit breaker behavior after the first call, check the logs again. The logs indicate a state transition from Open to Half-Open upon receiving the first request, as shown below:

```
Event STATE_TRANSITION published: 2024-07-01T18:40:38.965423+02:00[Eu-
rope/Rome]: CircuitBreaker 'catalogServiceCircuitBreak-
er' changed state from OPEN to HALF_OPEN
```

To confirm the circuit breaker's behavior after consecutive successful calls, check the logs again. After the second successful call, the logs show a state transition from Half-Open to Closed, as illustrated below:

```
Event STATE_TRANSITION published: 2024-07-01T18:40:39.521805+02:00[Eu-
rope/Rome]: CircuitBreaker 'catalogServiceCircuitBreak-
er' changed state from HALF_OPEN to CLOSED
```

Well, we have made the Easyshop application more resilient with Spring Cloud Gateway and Resilience4J. All that remains is to make the system secure as well by implementing OIDC and OAuth2.

# OAuth2 client and resource servers with Spring Security

To secure the Easyshop application, we will implement the OIDC and OAuth2 protocols using Spring Security. Specifically, edge-service, the microservice that acts

as the API Gateway, will be the OAuth2 client, while the downstream microservices, such as catalog service, will act as the resource server.

Spring Security is the Spring module that handles security in both imperative and reactive ways. To implement security, Spring Security uses a chain of filters, which are invoked in a certain order, and which intercept calls to RestControllers. In the imperative version, filters are implemented with classes of the type Filter, while in the reactive version, with classes of the type WebFilter.

# Configuring the OAuth2 client on Keycloak

Before using Spring Security, we need to configure a client application on Keycloak, that is, the application that will act as the OAuth2 client, which, in our case, will be the edge service. We again use the container bash of Keycloak as we did before, using the following command:

```
docker exec -it easyshop-keycloak bash
```

After that, let us go to the path **`/opt/keycloak/bin`** using the command:

```
cd /opt/keycloak/bin/
```

Let us log in as an admin user using the following command:

```
./kcadm.sh config credentials --server http://localhost:8080 \
  --realm master --user user --password password
```

Now we can create the client application using Keycloak admin CLI, with the following command:

```
./kcadm.sh create clients -r Easyshop \
  -s clientId=edge-service -s enabled=true \
  -s publicClient=false -s secret=easyshop-secret \
  -s 'redirectUris= ["http://localhost:8090",
                     "http://localhost:8090/login/oauth2/code/*"]'
```

Running this command creates the client application, edge service, with clientId and client secret defined by us. Also, with the **`publicClient=false`** option, we are defining the application as non-public. Public applications can use the clientId without the client secret to receive the authorization code. Usually, public client applications are web or mobile applications, where it is not safe to maintain a secret. **`RedirectUris`** is used to define the URLs to which Keycloak should redirect the browser after the user logs in and logs out. In this case, the first URI refers to logout, and the second is for login. In fact, Spring Security uses the following URI:

**`http://localhost:8090/login/oauth2/code/<provider_name>`** to accept the authorization code of a configured provider.

Now that we have created the client application on the authorization server, all that remains is to configure the edge service to use OIDC and OAuth2.

# Configuring the OAuth2 client on Spring Security

To implement the OAuth2 client, we need to import into the edge service project, the OAuth2 client dependency. The Maven dependency to add is shown below:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

The OAuth client module already imports the core Spring Security dependency. In addition, it is agnostic to the provider used. In this chapter, we will use Keycloak, but you can easily switch to other providers.

First, we configure the client application information we used in Keycloak. In the **`application.yml`** file, we then add the following configuration:

```
spring:
  ...
  cloud:
    gateway:
      ...
  security:
    oauth2:
      client:
        registration:
          keycloak:
            provider: keycloak
            authorization-grant-type: authorization_code
            client-id: edge-service
            client-secret: easyshop-secret
            scope: openid
        provider:
          keycloak:
            issuer-uri: http://localhost:8091/realms/Easyshop
```

The **spring.security.oauth2.client.registration** property allows several client applications to be registered. In this case, we configured one, registering it with the name **keycloak**.

By specifying the **openid** scope, you are indicating to Spring Security that the OIDC 1.0 protocol should be used. Spring will then use OIDC-specific components, such as **OidcUserService**, instead of the default **DefaultAuth2UserService**.

With the configuration provided, two endpoints are implicitly configured:

- The **/oauth2/authorization/keycloak** endpoint that allows the browser to redirect to the authorization server login page for unauthenticated calls.
- The **/login/oauth2/code/keycloak** endpoint that is used by the authorization server to redirect the browser after successful login by the user. The endpoint will be enriched by the authorization server with the query parameter code, which will then be swapped to obtain the ID token and access token by invoking the endpoint token.

The issuer-uri allows a configuration endpoint or authorization server metadata endpoint to be specified, so that other URIs, such as the **token** endpoint, **userinfo** endpoint, **jwks** endpoint, can be retrieved. For each registered provider, an instance of **ClientRegistration** is created, used by a bean of type **ClientRegistrationRepository**, which contains all registered providers (such as contains a list of **ClientRegistration**). By default, the **InMemoryClientRegistrationRepository** class is used, which stores in memory the list of **ClientRegistration**.

In addition, we add among the default edge service filters, the **SaveSession** filter, which forces the session to be saved before forwarding the call to the downstream services. The configuration is shown below:

```
spring:
  ...
  cloud:
    gateway:
      default-filters:
        - name: Retry
          ...
        - SaveSession
```

Let us now customize the configuration of the Spring Security chain. Let us create the security package and write the **SecurityConfig** class:

```
@Configuration
@EnableWebFluxSecurity
```

```
public class SecurityConfig {

    @Bean
    SecurityWebFilterChain springSecurityFilterChain(
            ServerHttpSecurity http,
            ReactiveClientRegistrationRepository clientRegistrationRepository) {

        return http
                .authorizeExchange(exchange -> exchange
                      .pathMatchers("/").permitAll()
                      .anyExchange().authenticated())
                .oauth2Login(Customizer.withDefaults())
                .logout(logout -> logout
                    .logoutSuccessHandler(
                          oidcLogoutSuccessHandler(clientRegistrationRepository)))
                .csrf(ServerHttpSecurity.CsrfSpec::disable)
                .build();
    }

    private ServerLogoutSuccessHandler oidcLogoutSuccessHandler(
            ReactiveClientRegistrationRepository clientRegistrationRepository) {
        var oidcLogoutSuccessHandler =
                new OidcClientInitiatedServerLogoutSuccessHandler(
                                              clientRegistrationRepository);
        oidcLogoutSuccessHandler.setPostLogoutRedirectUri("{baseUrl}");
        return oidcLogoutSuccessHandler;
    }
}
```

To define the Spring Security reactive chain, an instance of type **`SecurityWebFilterChain`** is used, which is created from an object of type **`ServerHttpSecurity`**. The object of type **`ServerHttpSecurity`** allows configuring aspects of security for specific HTTP requests. Let us analyze the following methods of the chain:

- The **`authorizeExchange`** method allows you to specify permissions on specific requests, based on parsed path patterns. In this case, for any endpoint, the user can access it only if they are authenticated, except / endpoint, so that logout can lead to an eventual home page of a single page application.

- The **`oauth2Login`** method allows us to configure authentication using OIDC. In this case, we do not customize any configuration.
- The **`logout`** method allows us to define a handler called after the user logs out. In this case, the handler of type **`OidcClientInitiatedServerLogoutSuccessHandler`** is used, which allows not only to clear the Spring Security session, but also to call the authorization server's logout endpoint to delete the token and session on the IS. The logout endpoint in Spring Security is **`/logout`**, with the POST method.

The **`csrf`** method allows us to specify a strategy for protection towards the Cross-Site Request Forgery (CSRF). We will disable the protection at this time for simplicity, but we will see later how to enable it.

We finished the configuration of Spring Security on edge service. Let us create an API that allows us to retrieve the logged-in user's data. In the **`api`** package, we write the following **`UserController`** class:

```
@RestController
@RequestMapping("user")
public class UserController {

    @GetMapping
    public User whoIam(@AuthenticationPrincipal OidcUser oidcUser) {
        return User.builder()
                .username(oidcUser.getPreferredUsername())
                .email(oidcUser.getEmail())
                .firstName(oidcUser.getGivenName())
                .lastName(oidcUser.getFamilyName())
                .roles(List.of("manager"))
                .build();
    }

    @Builder
    public record User(String username,
                       String email,
                       String firstName,
                       String lastName,
                       List<String> roles){}
}
```

As mentioned above, by specifying the **`openid`** scope, Spring Security uses the OIDC components, including the **`OidcUser`** class, which contains information about the logged-in user. Also, currently, the value of roles is hardcoded, but in the next section, we will see how to retrieve the configured user roles on Keycloak.

We start the edge service and open a browser tab. It is recommended to always open an incognito window to avoid cache and cookie problems. Enter the URL to retrieve the user's information **`http://localhost:8090/user`**

Edge service, receiving an unauthenticated call, performs a redirect to the Keycloak login page, as shown in *Figure 7.5:*

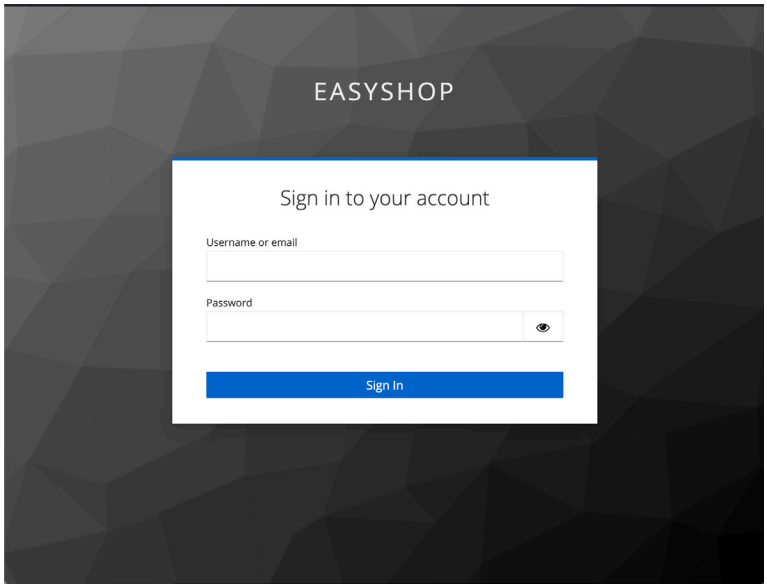



***Figure 7.5:*** *When the is not authenticated, edge service redirects the browser to the Keycloak logic page*

We enter the credentials of the user John (username as john, password as john). After logging in, you should see the following result on the browser page:

```
{
    "username": "john",
    "email": "john@mail.com",
    "firstName": "John",
    "lastName": "Smith",
    "roles": [
        "manager"
    ]
}
```

Well, we were able to successfully consume the user API after authenticating ourselves. Edge service behind the scenes made the HTTP calls described in *Figure 7.6:*

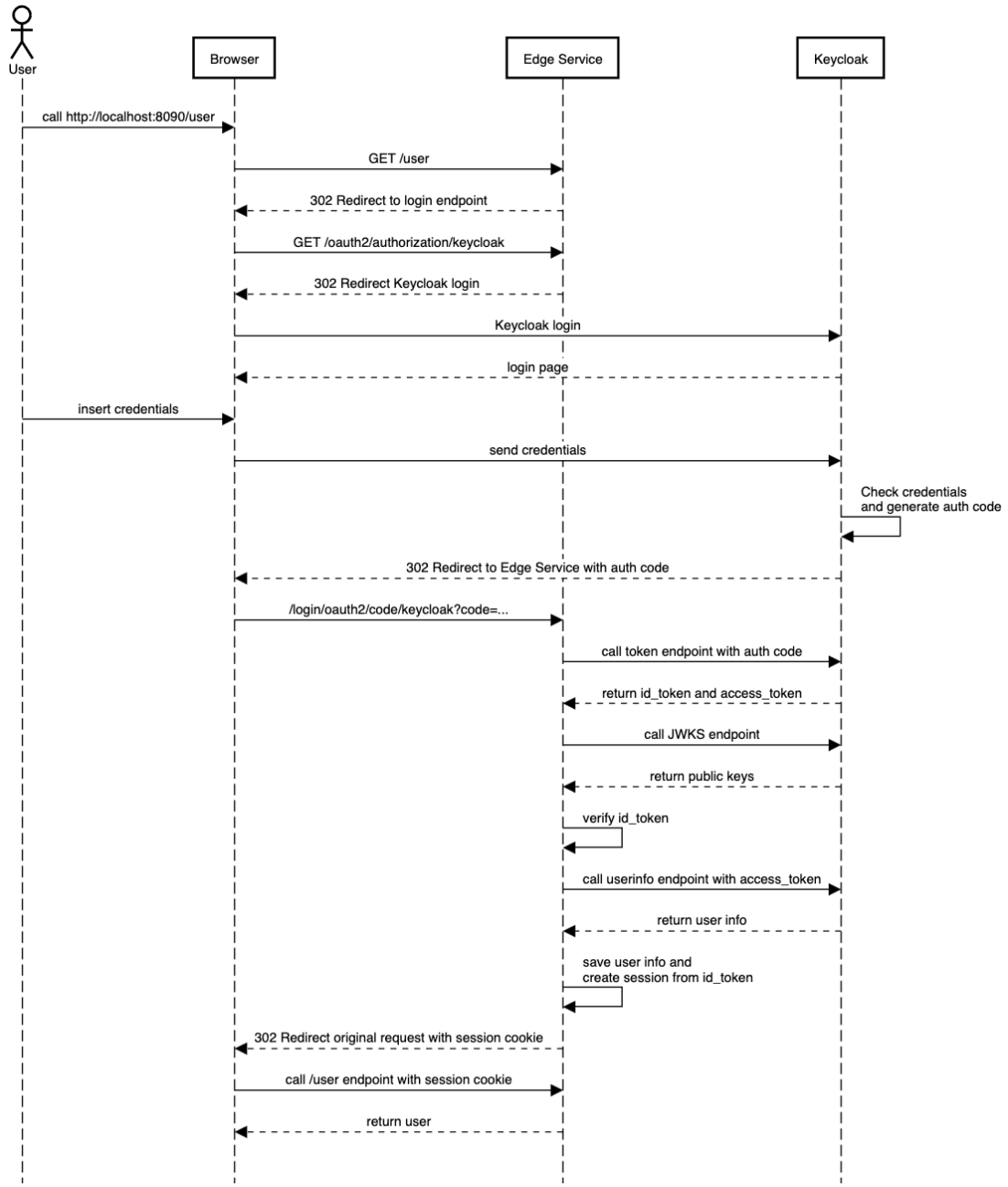



***Figure 7.6:*** *Calling the user endpoint for an unauthenticated user*

You can verify calls made from the browser by enabling the Browser console, and going to the Network tab, while you can verify calls received and made from edge service by enabling TRACE HTTP logs, as we have already seen in previous chapters:

```
logging:
  level:
    org.springframework.web: TRACE
```

**Note: The edge service only verifies the token ID, because it is not its job to also verify the access token. In addition, the User Info endpoint is also invoked to enrich the logged-in user information present in the ID token claims.**

Try making the call to the user endpoint again from the same incognito browser window. You will see from the edge service logs that because the request is passed with a valid session cookie, the edge service returns the endpoint result directly without going through the authentication flow. However, if you restart the edge service and make the call again from the browser, you will see that, although the request has the cookie, the microservice re-executes the authentication flow because after it is turned off, it has lost all the sessions saved in memory. Currently, edge service is not stateless. To make it so, we would have to delegate session saving to a tool, such as a database or a distributed cache.

## Spring Session with Redis to manage sessions

Spring Session is a module of Spring that provides APIs to manage user-session information. In addition to its core module, Spring Session Core, Spring Session provides the following modules for out-of-box integration with some providers:

Spring Session Data Redis: Provides integration with Redis.

Spring Session JDBC: Provides integration with a relational database.

Spring Session Hazelcast: Provides integration with Hazelcast.

Spring Session MongoDB: Provides integration with MongoDB.

To manage HTTP sessions in Easyshop we will use Redis, which is an in-memory database that can be used as a distributed cache on multiple nodes.

To start Redis locally, we will use Docker. We then add the Redis service in the **`docker-compose.yml`** file:

```
  redis:
    image: "redis:7.2"
    container_name: "easyshop-redis"
    ports:
      - 6379:6379
```

Integrating Redis with Spring is very simple: just import the specialized **Spring Session** dependency for Redis and **Spring Data Redis Reactive**, to use the reactive module (since edge service uses WebFlux):

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-data-redis</artifactId>
</dependency>
```

Having imported the dependencies, we have to do is configure the Redis cluster host and port, in the **`application.yml`** file:

```
spring:
  ...
  data:
    redis:
      host: localhost
      port: 6379
```

In this case, we could have also avoided setting the properties above, as their value is the same as the default configurations.

Inside the **`infra/docker`** folder, we run the **`docker compose up -d`** command, so that we also start the Redis container, after which we start the edge service. Open an incognito browser tab and invoke the URL **`http://localhosty:8090/user`**. Enter user John's credentials, after which, once the user John's information is displayed on the browser page, look at the edge service logs. As before, the edge service makes several HTTP calls to implement the OIDC flow. Try restarting the edge service and calling the same URL again from the same browser window. You will see that this time, no HTTP call is made, as the edge service retrieves the session from Redis and recognizes the authenticated user.

Thanks to Spring Session, we have untied the edge microservice from handling session storage. All that remains is to implement the resource server on the catalog microservice.

# Configuring the resource server on Spring Security

The resource server verifies whether the authenticated user is authorized to consume the

enriched with the access token in the header, which is validated through a public key retrieved from the JWKS endpoint. In fact, as mentioned at the beginning of this chapter, Spring Security caches the list of public keys returned by the JWKS endpoint so that the IS is not overly stressed.

Before moving on to the resource server (catalog service), we need to make some minor changes so that the edge service handles the access token optimally.

## Access token management in the OAuth2 client

The first step required to make catalog service receive the access token, is to add to the edge service the **`TokenRelay`** filter, which allows precisely to forward to downstream services the access token into the following **`Authorization`** header:

```
spring:
  ...
  cloud:
    gateway:
      default-filters:
        - name: Retry
          ...
        - SaveSession
        - TokenRelay
```

However, by default, Spring Security stores access tokens in memory, which makes edge service not yet completely stateless.

To make it so, we can store the access token in the session, which is already stored in Redis. Spring Security wraps authenticated user information, including the access token, in objects of type **`OAuth2AuthorizedClient`**:

```
public class OAuth2AuthorizedClient implements Serializable {


 private final ClientRegistration clientRegistration;
 private final String principalName;
 private final OAuth2AccessToken accessToken;
 private final OAuth2RefreshToken refreshToken;
...
}
```

OAuth2AuthorizedClient instances are stored, retrieved and deleted through an instance of type **`ServerOAuth2AuthorizedClientRepository`** which is an interface. By default, the implementation that stores **`OAuth2AuthorizedClient`**

Fortunately, Spring also provides the **`WebSessionServerOAuth2AuthorizedClientRepository`** implementation that manages **`OAuth2AuthorizedClient`** instances via sessions. To use this implementation, we define the **`ServerOAuth2AuthorizedClientRepository`** bean in the following **`SecurityConfig`** class:

```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {

    ....

    @Bean
    ServerOAuth2AuthorizedClientRepository serverOAuth2AuthorizedClien-
tRepository() {
        return new WebSessionServerOAuth2AuthorizedClientRepository();
    }
}
```

Now that edge service is fully stateless and handles access tokens optimally. All that remains is to implement the resource server on the catalog service.

## Resource server implementation

To make the catalog service act as a resource server, we import the following **Spring OAuth2 Resource Server** dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

We configure, in the **`application.properties`** file, the only property the resource server needs, which is the value of the issuer endpoint:

```
spring.security.oauth2.resourceserver.jwt.issuer-uri=http://localhost:8091/
realms/Easyshop
```

Other useful authorization endpoints, such as the one for JWKS, can be retrieved from this endpoint. The last step is to customize the Spring Security chain so that it tells Spring that the application should act as a resource server. We create the security package and add the following **`SecurityConfig`** class:

```
@Configuration
@EnableWebFluxSecurity
```

```
public class SecurityConfig {

    private static final String PRODUCT_PATH = "/products/**";
    private static final String MANAGER_ROLE = "manager";

    @Bean
    SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        return http
                .authorizeExchange(exchange -> exchange
                        .anyExchange().authenticated())
                .oauth2ResourceServer(oauth2Conf -> oauth2Conf
                        .jwt(Customizer.withDefaults()))
                .requestCache(requestCacheSpec -> requestCacheSpec
                        .requestCache(NoOpServerRequestCache.getInstance()))
                .csrf(ServerHttpSecurity.CsrfSpec::disable)
                .build();
    }
}
```

The **`oauth2ResourceServer`** method allows us to indicate to Spring that our application acts as a resource server. Finally, the **`requestCache`** method allows to specify session management. Since the catalog service does not need to manage sessions because it consumes the JWT, with the static method **`NoOpServerRequestCache.getInstance()`** we indicate to Spring that it should not create sessions. Also, in this case, it is also right to disable protection on CSRF attacks since our application is not subject to them by not using cookies.

We start the catalog service and, from the browser, invoke the **`findProductByCode`** API via edge service **`http://localhost:8090/products/LAP-ABCD`**. After logging in with the user John, you will see the following response payload from the catalog service:

```
{
   "code":"LAP-ABCD",
   "name":"ALaptop",
   "category":"laptop",
   "price":30000,
   "brand":"FirstBrand"
}
```

The integration of the OAuth2 resource server with Spring Security is simple. However, we can do better than that. We can make it so that only users with a **manager** role can consume the write API, while all other users (with a **user** role), can consume only the read API.

# Role based access control with Spring Security

RBAC is a technique for assigning permissions to users based on their assigned roles.

In Keycloak, we have already censored the user John with the role of manager and the user Mrossi with the role of user. However, we should make some changes so that the role information is present among the claims of both the ID token and the access token.

## Roles claim in the token ID

To make the role information available in the claims of both ID token and access token, we access the Keycloak console from the browser by typing the URL `http://localhost:8091`. We enter the username and password of the admin (user or password) and, from the dropdown menu, choose the **Easyshop** realm, as shown in *Figure 7.7*:

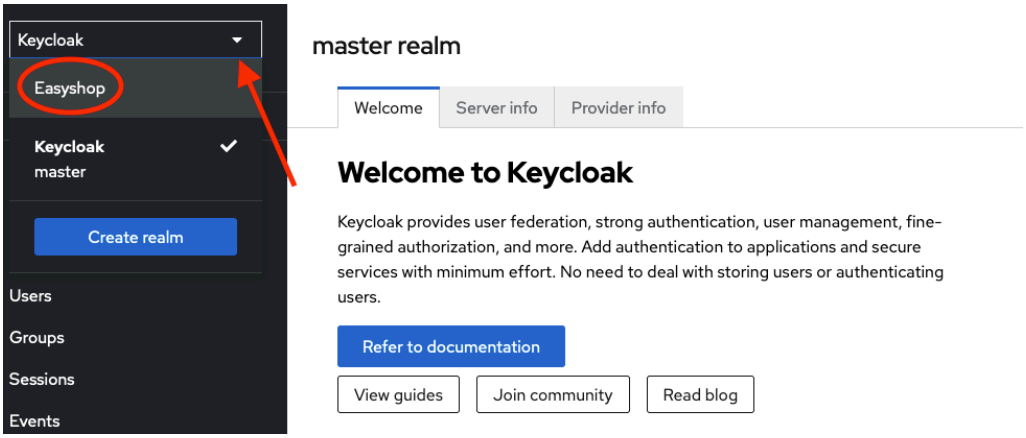



***Figure 7.7:*** *Keycloak's dropdown menu that allows you to choose the realm*

Select Client scopes from the menu on the left. On this page, you will see all scopes defined on Keycloak, and you can also create new ones. We are interested in editing **roles** claims, so click on the latter. Let us click on the **Mappers** tab, which allows you to view a set of claims and edit their configuration. We are interested in the **realm roles** mapper, so we click on the latter. This mapper already maps the list of logged-in user roles to an access token claim called **realm_roles**. We changed the name of the claim so that it is called **roles** and also included it in the ID token, since we have created the user API on edge service that returns information about the user. Edge service reads the claims from the ID token, while catalog service is from the access token. *Figure 7.8* shows the final changes on the realm roles mapper:

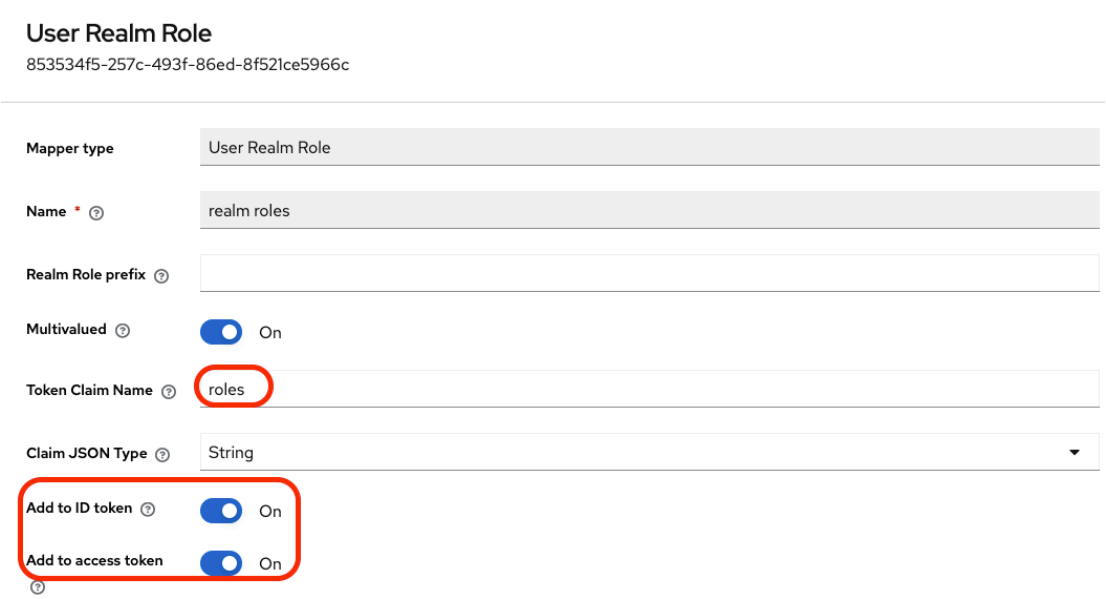



*Figure 7.8: Mapping roles into the claim roles, which will be present in both the ID token and the access token*

With this configuration, the edge service will be able to know the roles of the logged-in user using the value of the roles claim within the token ID. We modify the method of the **`UserController`** class so that it uses the roles returned by the ID token:

```
@RestController
@RequestMapping("user")
public class UserController {

    @GetMapping
    public User whoIam(@AuthenticationPrincipal OidcUser oidcUser) {
        return User.builder()
                .username(oidcUser.getPreferredUsername())
                .email(oidcUser.getEmail())
                .firstName(oidcUser.getGivenName())
                .lastName(oidcUser.getFamilyName())
                .roles(oidcUser.getClaimAsStringList("roles"))
                .build();
    }
   ...
}
```

Try invoking the user API from the browser with user John, and you will see that a list of roles will be presented in the roles field that includes that of manager (as well as other roles added by default by Keycloak):

```
{
   "username":"john",
   "email":"john@mail.com",
   "firstName":"John",
   "lastName":"Smith",
   "roles":[
      "default-roles-easyshop",
      "manager",
      "offline_access",
      "uma_authorization"
   ]
}
```

We have included the role information of the logged-in user in the OAuth2 client so that it can be used by the API user. However, reading the role is crucial information for the resource server, since it must apply security rules on the latter.

## Protecting APIs with RBAC in the resource server

Spring Security extracts, by default, roles from scope and scp claims. Since the claim name of the roles, we configured in Keycloak is roles, we need to make some minor changes.

Spring uses the **`JwtGrantedAuthoritiesConverter`** class, to extrapolate roles from the JWT. This class creates a list of **`GrantedAuthorities`**, based on the extrapolated list of roles. This converter in turn is used by an instance of **`ReactiveJwtAuthenticationConverter`**, which wraps the authenticated user's information, including roles, in a class of type **`JwtAuthenticationToken`** (which is the counterpart of the edge service class **`OidcUser`**).

To customize the two converters so that the roles are extracted from the roles claim, we need to define a bean of type **`ReactiveJwtAuthenticationConverter`**. The following code demonstrates how to configure this bean in the **`SecurityConfig`** class of catalog service:

```
@Bean
ReactiveJwtAuthenticationConverter jwtAuthenticationConverter() {
    var jwtGrantedAuthoritiesConverter = new JwtGrantedAuthoritiesConverter();
    jwtGrantedAuthoritiesConverter.setAuthorityPrefix("ROLE_");
```

```
    var jwtAuthenticationConverter = new ReactiveJwtAuthenticationConverter();
    jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(
            new ReactiveJwtGrantedAuthoritiesConverterAdapter(jwtGrantedAu-
thoritiesConverter));
    return jwtAuthenticationConverter;
}
```

In addition to defining the claim to retrieve roles, with the **`setAuthoritiesClaimName`** method, we also defined the role prefix that will be mapped to the **`GrantedAuthority`** class, with the **`setAuthorityPrefix`** method (the default value is **`SCOPE_`**).

To restrict access to the write APIs so that only users with the **manager** role can invoke them, we need to modify the Spring Security filter chain. The following code demonstrates how to configure the **`SecurityConfig`** class to enforce these restrictions:

```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {

    private static final String PRODUCT_PATH = "/products/**";
    private static final String MANAGER_ROLE = "manager";

    @Bean
    SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http) {
        return http
                .authorizeExchange(exchange -> exchange
                        .pathMatchers(HttpMethod.POST, PRODUCT_PATH)
                            .hasAnyRole(MANAGER_ROLE)
                        .pathMatchers(HttpMethod.PUT, PRODUCT_PATH)
                            .hasAnyRole(MANAGER_ROLE)
                        .pathMatchers(HttpMethod.DELETE, PRODUCT_PATH)
                            .hasAnyRole(MANAGER_ROLE)
                        .anyExchange().authenticated())
                .oauth2ResourceServer(oauth2Conf ->
                        oauth2Conf.jwt(Customizer.withDefaults()))
                .requestCache(requestCacheSpec ->
                        requestCacheSpec
                               .requestCache(NoOpServerRequestCache.getInstance()))
                .build();
```

```
    }
    ...
}
```

The **`pathMatchers`** method allows a parsed path pattern to be defined to match endpoints, while the **`hasAnyRole`** method allows a list of roles (without the **`ROLE_`** prefix) required to consume the same endpoints to be associated with them. The **`hasAnyAuthority`** method can also be used, but in that case, the role prefix (in our case, **`ROLE_manager`**) must also be included.

The approach demonstrated here applies RBAC at the request level. Alternatively, RBAC can also be implemented at the Java method level by leveraging the **`@PreAuthorize`** annotation. For example, **`@PreAuthorize("hasAnyRole('MANAGER_ROLE')")`** can be used to enforce role-based restrictions on specific methods. To enable method-level security, the **`@EnableMethodSecurity`** annotation must be added to a class annotated with **`@Configuration`**. For further details, refer to the official documentation: **https://docs.spring.io/spring-security/reference/servlet/authorization/method-security.html**

To test how the roles work, we can directly invoke the catalog service API with the access token in the header. To retrieve the JWT, you have several ways, for example, you can retrieve it from the catalog service logs, or we can use a tool like **Postman** that allows you to extract the access token automatically from an OAuth2 Authorization Code flow. Let us use the second mode. We start Postman and import the collection located in the **`infra/postman`** folder. We select the **`addProduct`** API and click at the bottom on **Get New Access Token**. Figure 7.9 shows the Postman screen affected:

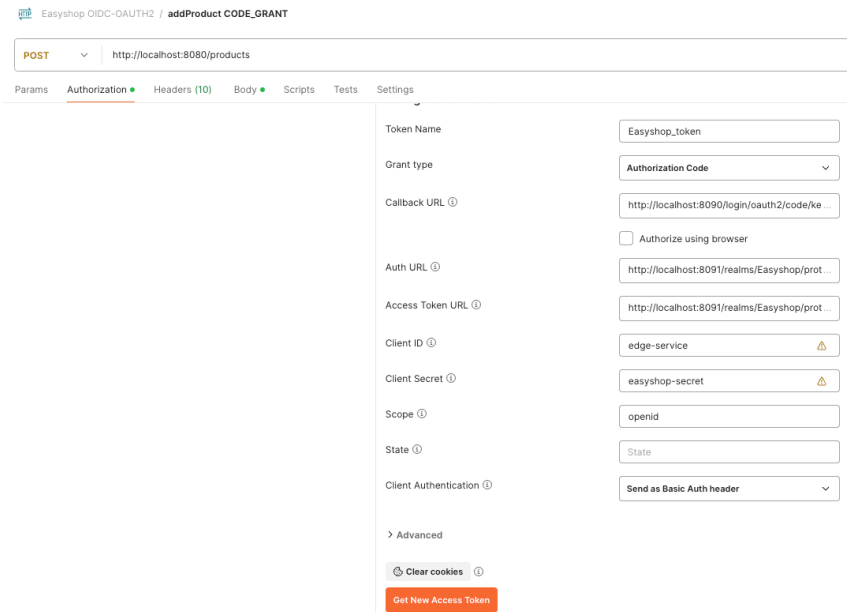



*Figure 7.9:*

A window will open showing the Keycloak login page. Let us authenticate with the user Mrossi. Once authenticated, we can click on the **Use Token** button, which allows us to automatically import the access token. At this point, we invoke the API by clicking on the **Send** button. We will correctly get a 403 forbidden, since the Mrossi user cannot invoke the POST API, as he does not have the manager role. At the bottom, we click on the **Clear cookie's** button and then again on **Get New Access Token**. This time, let us authenticate with the user john and invoke the API. We will correctly get a 201 created, since the user John has the manager role.

Well, we protected our APIs with OIDC and OAuth2, enforcing a policy on roles. However, even though we have programmatically tested the operation of the flow by directly invoking the API, it is important to write integration tests that certify that it works properly.

# Testing OIDC and OAuth2 flow with Spring Security

To test authentication with OIDC in edge service, we can use the user endpoint, writing integration tests with **`WebTestClient`**, already seen in chapter two, and the **`SecurityMockServerConfigurers`** class, which allows us to mock an OIDC login by providing either an ID token or an instance of **`OidcUser`**. To use the **`SecurityMockServerConfigurers`** class, we need to import the following Spring Security test dependency into the edge service:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

Inside the test folder, create the **`api`** package and write the following test class:

```
@WebFluxTest(UserController.class)
@Import(SecurityConfig.class)
class UserControllerTest {

    @Autowired
    WebTestClient webTestClient;

    @MockBean
    ReactiveClientRegistrationRepository repository;

    @Test
    void authenticatedUserTest() {
```

```
        var expectedUser = UserController.User.builder()
                .username("john")
                .firstName("John")
                .lastName("Smith")
                .email("john@mail.com")
                .roles(List.of("manager"))
                .build();

        webTestClient
                .mutateWith(mockIdToken(expectedUser))
                .get()
                .uri("/user")
                .exchange()
                .expectStatus().is2xxSuccessful()
                .expectBody(UserController.User.class)
                .value(user -> assertThat(user).isEqualTo(expectedUser));
    }

    private SecurityMockServerConfigurers.OidcLoginMutator
      mockIdToken(UserController.User expectedUser) {
        return SecurityMockServerConfigurers.mockOidcLogin()
                .idToken(builder ->
                        builder
                                .claim(StandardClaimNames.PREFERRED_USERNAME,
                                        expectedUser.username())
                                .claim(StandardClaimNames.GIVEN_NAME,
                                        expectedUser.firstName())
                                .claim(StandardClaimNames.FAMILY_NAME,
                                        expectedUser.lastName())
                                .claim(StandardClaimNames.EMAIL,
                                        expectedUser.email())
                                .claim("roles",
                                        expectedUser.roles())
                );
    }
}
```

The test enriches the **`webTestClient`** instance with the OIDC mock login using the **`mutateWith`** method. Run the test, and you will see that it is passed successfully. We leave it to you to write the test that verifies that for an unauthenticated user, a 302 Found is received (since an unauthenticated user is redirected to the login page).

Regarding the catalog service, we can apply the same approach used for the edge service. Specifically, we can use the **`SecurityMockServerConfigurers`** class to simulate the access token for testing purposes. The following code demonstrates how to configure a simulated access token with a specified role:

```
private SecurityMockServerConfigurers.
JwtMutator mockAccessToken(String role) {
    return SecurityMockServerConfigurers.mockJwt()
            .authorities(new SimpleGrantedAuthority(role));
}
```

You will have to modify the **`ProductControllerTest`** class tests by adding the **`mockAccessToken`** method.

# Configuring OAuth2Client for SPA

We have concluded the securing part of Easyshop. However, for completeness, it is only fair to understand what changes to make on the edge service to best integrate it with a **Single Page Application (SPA)**, although integration with the latter is not the focus of this book.

In our authentication flow, the edge service, after the user has successfully authenticated, creates a session by associating it with the ID token received from the authorization server and returns the session information in the form of a cookie to the browser. Each HTTP request made by the browser will contain the session cookie, which will be validated by the edge service. However, cookies are vulnerable to CSRF attacks for requests that change the state of the resource (POST, PUT, PATCH, and DELETE).

To protect the application from CSRF attacks, Spring Security generates a CSRF token that is provided to the client at the beginning of the session. The client must provide the token for each request that changes the state of the resource. By default, the CSRF token is stored in the session, but Spring provides a way to also store it in cookies. The edge service security chain becomes as follows:

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(
        ServerHttpSecurity http,
        ReactiveClientRegistrationRepository clientRegistrationRepository) {
    return http
```

```
            .authorizeExchange(exchange -> exchange
                    .anyExchange().authenticated())
            .oauth2Login(Customizer.withDefaults())
            .csrf(csrf -> csrf
                    .csrfTokenRepository(CookieServerCsrfTokenRepository
                            .withHttpOnlyFalse())
            )
            .build();
}

...

@Bean
WebFilter csrfWebFilter() {
    return (exchange, chain) -> {
        exchange.getResponse().beforeCommit(() ->
                Mono.defer(() -> {
                    Mono<CsrfToken> csrfToken = exchange
                            .getAttribute(CsrfToken.class.getName());
                    return csrfToken != null ? csrfToken.then() : Mono.empty();
                }));
        return chain.filter(exchange);
    };
}
```

**Note: We also need to create a filter because of a bug in the reactive version of the CSRF filter (https://github.com/spring-projects/spring-security/issues/5766).**

Another aspect to consider when integrating with SPA is that accessing an API without first being authenticated should not lead to automatic redirection to the login page of the authorization server, but rather should return an unauthorized 401. It is the SPA that must handle the redirection to the login. To change this behavior in the edge service, we need to handle the exception thrown when an endpoint is invoked by an unauthenticated user, which, by default, leads to the login page. The final security chain becomes the following:

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(
        ServerHttpSecurity http,
        ReactiveClientRegistrationRepository clientRegistrationRepository) {
```

```
    return http
            .authorizeExchange(exchange -> exchange
                    .pathMatchers("/", "/*.css", "/*.js", "/favicon.ico")
                    .permitAll()
                    .anyExchange().authenticated())
            .exceptionHandling(exceptionHandling -> exceptionHandling
                    .authenticationEntryPoint(
                            new HttpStatusServerEntryPoint(HttpStatus
                                    .UNAUTHORIZED))
            )
            .oauth2Login(Customizer.withDefaults())
            .csrf(csrf -> csrf
                    .csrfTokenRepository(CookieServerCsrfTokenRepository
                            .withHttpOnlyFalse())
            )
            .build();
}
```

In addition, we have added static resources such as CSS and JavaScript files among the unprotected paths to make the SPA elements more usable.

# Conclusion

In this chapter, we looked at how to secure the Easyshop API. We created a new microservice, edge service, which acts as an API Gateway, in such a way as to handle cross-cutting concerns such as security aspects and resilience of Easyshop, applying retry and circuit breaker patterns. We have seen how the OIDC authentication flow and OAuth2 authorization flow work, implementing the authorization code flow with Spring Security and using Keycloak as the IS. In addition, we also applied a role policy to the resource server API so that only users with the manager role could use the catalog service writing API. However, to have an application ready to be released into production, we must also consider a good observability mode so that we can detect problems and possibly create alerts if unanticipated situations occur.

In the next chapter, we will see how to implement this with Spring and OpenTelemetry.

# Points to remember

- Spring Security uses a chain of filters to protect endpoints.
- OIDC is an authentication protocol based on OAuth2.
- OAuth 2.0 is the industry-standard protocol for authorization.

# Exercises

1. Implement the remaining jUnits.
2. Protect the order service API.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

**https://discord.bpbonline.com**



# Chapter 8
# Observability and Monitoring

## Introduction

In the world of microservices and distributed architectures, the ability to observe and monitor the behavior of applications is crucial to ensuring their reliability, availability, and performance. *Chapter 8,* entitled *Observability and Monitoring*, provides an overview of the techniques and tools used to improve observability and monitoring of modern applications. Starting from the releases of Spring Framework 6 and Spring Boot 3, Spring Boot Actuator, which is Spring's module focused on observability, now utilizes the Micrometer library. This library provides a vendor-neutral API for instrumenting code. In this chapter, we will enrich the edge service and catalog service microservices, adding Spring Boot Actuator to instrument metrics and distributed tracing. We will also use open-source observability tools to exploit these data.

In the first part, we will explore how to use *Grafana Loki*, a highly scalable log aggregation system, in our microservices. In the second part, we will explore *Spring Boot Actuator*, with which a variety of application-related metrics can be exposed. In this section, we explore how to integrate Spring Boot Actuator with *Prometheus*, an open-source monitoring system designed to collect and analyze large-scale metrics.

In part three, we will explore the use of *Micrometer Tracing* and *OpenTelemetry* to instrument tracing in Spring Boot applications. In addition, we will send traces to *Grafana Tempo*, a backend that handles tracing on Grafana.

Finally, in the last part of the chapter, we will use *Grafana* to visualize metrics and traces through dashboards. Grafana was chosen for this chapter because it provides a unified platform for observability when combined with its complementary tools such as Loki (for logs), Prometheus (for metrics), and Tempo (for tracing). Compared to other monitoring and logging tools like Splunk, Grafana offers a lightweight, open-source alternative with native support for different data sources and real-time visualization. *Figure 8.1* shows an overview of the tools we will use.

# Structure

In this chapter we will discuss the following topics:

- Log aggregation with Grafana Loki
- Metrics with Spring Boot Actuator and Prometheus
- Tracing with Micrometer, OpenTelemetry, and Grafana Tempo
- Monitoring with Grafana

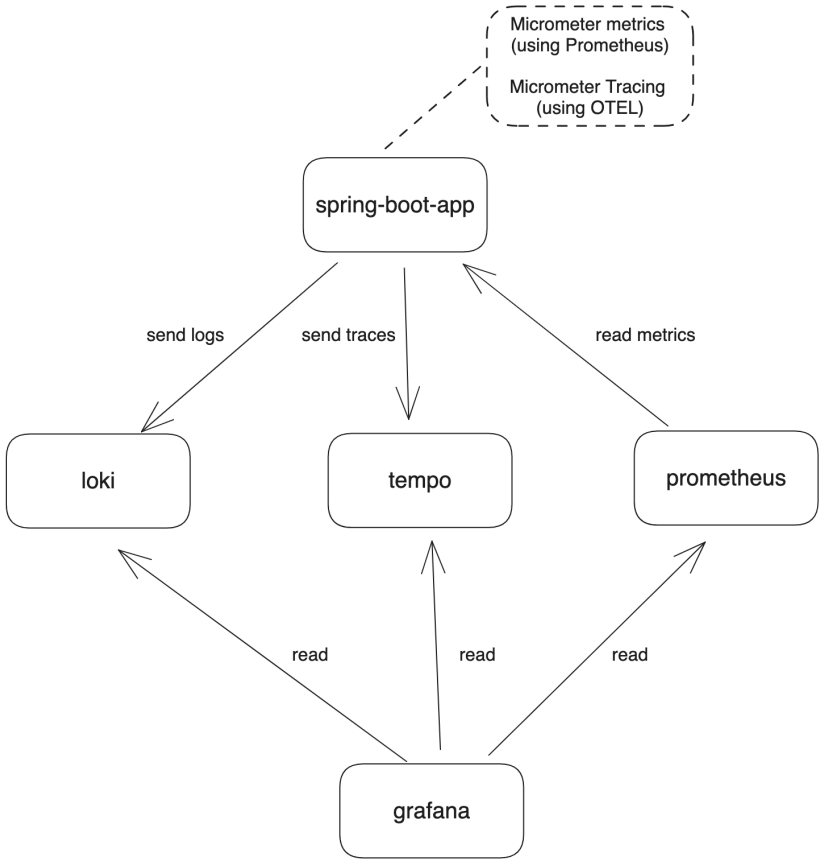



***Figure 8.1**: Observability tools that we will use in this chapter*

# Objectives

By the end of this chapter, you will have gained practical skills in configuring and using observability and monitoring tools. You will be able to implement, with Spring Boot Actuator, comprehensive solutions for log management, metrics monitoring, and distributed tracing in their applications, using open-source tools from Loki, Tempo, and Grafana, providing greater operational visibility and thereby improving the ability to respond to incidents promptly. This chapter will provide the foundation needed to build applications that will be ready to go into production.

# Log aggregation with Grafana Loki

Logs are chronological records of activities and events occurring within an application. These records include informational messages, warnings, errors, and debugs generated by the application during its operation. Logs are essential for application diagnostics, monitoring, maintenance, and improvement because they provide a detailed record of system operations and behaviors.

## Logging in Spring Boot

Spring Boot facilitates the implementation and management of logs through simple, integrated configuration. It uses **Simple Logging Facade for Java** (**SLF4J**) APIs that can be implemented out-of-box by the logging libraries of **Logback**, **Log4j2**, and **Java Util Logging** (**JUL**). Spring Boot uses **Logback** as the default logging implementation. No additional configuration is required to start using logs when importing a Spring Boot starter.

The default format of the logs is as follows:

```
<timestamp> <log_level> <process_id> --- <app_name> <thread_name>
<correlation_id> <logger_name> <message>
```

Let us look at each field in detail:

- **Timestamp**: It is the date and time in milliseconds of logging.
- **log_level:** The log level. It can be ERROR, WARN, INFO, DEBUG e TRACE. Logback does not have the FATAL log level.
- **process_id:** This field contains the unique identifier of the process that generated the log.
- **Separator(---):** This is a fixed separator used to improve the readability of the log format.
- **app_name:** The name of the application, if the spring.application.name property is set.

- **thread_name:** The name of the thread.
- **correlation_id:** A correlation ID, if tracing is enabled.
- **logger_name:** The name of the Logger, which is usually the class that writes the log.
- **Message:** It is the log message.

By default, messages with log level ERROR, WARN and INFO are logged. You can, however, change the visibility of logs for certain packages, using the property **`logging.level.<logger_name>`**, as shown in the following example:

```
logging.level.root=warn

logging.level.org.hibernate=error
```

With the first property, we set the generic visibility for all logs. In general, it is the logs with WARN and ERROR levels will be printed. With the second property, we go to configure the visibility of classes-specific logs that are in the **`org.hibernate`** package and subpackages. For these logs, only messages with an ERROR level will be printed.

Spring Boot simplifies the process of creating log groups, allowing developers to apply common configurations to multiple loggers. For example, the following code snippet demonstrates how to group and configure logging levels for Tomcat-related loggers:

```
logging.group.tomcat=org.apache.catalina, org.apache.coyote, org.apache.
tomcat

logging.level.tomcat=TRACE
```

If you want to use Logback-specific configurations, you can add the **`logaback.xml`** file to the classpath. If you want to use the Logback extensions, you can add the **`logback-spring.xml`** file to the classpath.

# Working with Loki

In traditional applications, logs are printed in text files, and stored on the machine hosting the application. In Cloud Native applications, log management must necessarily be different. Complete logs of an HTTP request may involve multiple microservices, and applications are replicated to apply horizontal scaling. In Cloud Native applications, logs must be printed in standard output. It is delegated to external tools to capture logs from standard output, store them, aggregate them, filter them, and make them available for analysis.

Grafana Loki is a log aggregation system designed to be highly scalable and easy to integrate with Cloud Native applications. Loki receives logs via HTTP/HTTPS protocol, and indexes and stores them, and allows users to query specific logs using LogQL, Loki's powerful query language.

In order to start Loki locally, we use the Docker container, adding the following piece of code in the **`docker-compose.yml`** file:

```
loki:
 image: grafana/loki:2.9.4
 container_name: "easyshop-loki"
 extra_hosts: ['host.docker.internal:host-gateway']
 command: [ "-config.file=/etc/loki/local-config.yaml" ]
 ports:
  - "3100:3100"
```

**Note: In the chapter-08 folder, you will find the `docker-compose.yml` file complete with all the services dedicated to observability and monitoring.**

For simplicity, the edge service will use the Logback appender for Loki, **`Loki4jAppender`**, which allows logs to be forwarded to Loki using the HTTP protocol. In a real production context, if the microservice is containerized, it might be a more viable choice to go for a tool like *Fluent Bit*, which captures logs from standard container output and sends them to Loki. You can configure the appender in the **`logback-spring.xml`** file.

In edge service, we create the file **`logback-spring.xml`**, inside the resources folder, as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/base.xml" />
    <springProperty scope="context" name="appName"
                    source="spring.application.name"/>

    <springProperty scope="context" name="lokiEndpoint" source="loki.endpoint"/>

    <appender name="LOKI" class="com.github.loki4j.logback.Loki4jAppender">
        <http>
            <url>${lokiEndpoint}/loki/api/v1/push</url>
        </http>
        <format>
            <label>
                <pattern>
  service_name=${appName},host=${HOSTNAME},traceID=%X{traceId:-NONE},level=%level
```

```
                </pattern>
            </label>
            <message>
                <pattern>${FILE_LOG_PATTERN}</pattern>
            </message>
            <sortByTime>true</sortByTime>
        </format>
    </appender>

    <root level="INFO">
        <appender-ref ref="LOKI"/>
    </root>
</configuration>
```

In the **`logback-spring.xml`** file, you can easily import properties from the **`application.properties`** (or **`application.yml`**) file using the **`springProperty`** tag. Let us go define the **`loki.endpoint`** variable in the edge-service **`application.yml`** file:

```
loki:
 endpoint: http://localhost:3100
```

With this simple configuration, the edge service will send logs to Loki. We will leave it to you to write the same configuration for the catalog service.

# Metrics with Spring Boot Actuator and Prometheus

In the context of microservice architectures, monitoring metrics is critical to ensure the health, performance, and availability of applications. Microservices, by their nature, are distributed systems that interact with each other across a network. This complexity makes it difficult to monitor and manage the behavior of the entire system without proper observability tools. Monitoring metrics provide critical information about application performance (e.g., API error rates, throughput, and latency measurements), enabling developers and operators to identify bottlenecks (e.g., high latency or excessive resource consumption), diagnose problems, and make data-driven decisions to optimize resources and improve the user experience. By analyzing resource utilization metrics, such as CPU, memory, and I/O, application performance can be optimized, and services can be ensured to respond efficiently to requests. The metrics also help determine when services need to scale horizontally or vertically to handle varying workloads, ensuring that resources are used optimally. Additionally, monitoring metrics and alerting play a vital role in

**Site Reliability Engineering (SRE)**, as they help ensure system reliability, availability, and performance by proactively identifying and addressing potential issues before they impact users.

Spring Boot Actuator is a Spring Boot module that provides monitoring and management capabilities for ready-to-use applications. The actuator exposes a set of HTTP endpoints that provide access to internal application information, such as metrics, health status, system information, and configurations. In order to collect metrics, it uses the Micrometer library (**https://micrometer.io**), which provides a vendor-neutral facade so these can be exported using a variety of formats, such as Prometheus and Datalog. For our application, we will use Prometheus, an open-source monitoring solution that stores metrics, just as Loki stores logs.

In order to start Prometheus locally, we add the following piece of code to the **`docker-compose.yml`** file:

```
prometheus:
 image: prom/prometheus:v2.50.0
 container_name: "easyshop-prometheus"
 extra_hosts: ['host.docker.internal:host-gateway']
 command:
  - --enable-feature=exemplar-storage
  - --config.file=/etc/prometheus/prometheus.yml
 volumes:
  - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml:ro
 ports:
  - "9095:9090"
```

In the **`infra/docker/prometheus`** folder, you will find the dedicated Prometheus configuration file that is used as the volume.

In order to use Spring Boot Actuator with Prometheus, we will import the following dependencies into the edge service:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
```

```
  <scope>runtime</scope>
</dependency>
```

Spring Boot Actuator provides several endpoints to monitor the application, for example, there is an endpoint that monitors the health of the microservice (**`/actuator/health`**), one dedicated to environment variables (**`/actuator/env`**), another to change the visibility of loggers at runtime (**`/actuator/loggers`**). For the full list of endpoints, visit the following official documentation:

**https://docs.spring.io/spring-boot/reference/actuator/endpoints.html**

However, because some of this information may contain sensitive data, by default only some endpoints are exposed. To enable all endpoints, we set the following property in the **`application.yml`** file:

```
management:
 endpoints:
  web:
   exposure:
    include: '*'
```

We also enable the publication of a histogram suitable for calculating aggregable percentile approximations (e.g., latency analysis, such as evaluating p90 and p99 of requests), enable detail on metrics, such as knowing disk space, and finally define a Micrometer tag to label all metrics by application name, allowing for categorization and filtering of metrics, an essential feature in multi-service environments:

```
management:
 endpoints:
  web:
   exposure:
    include: '*'
 endpoint:
  health:
   show-details: always
   show-components: always
 metrics:
  distribution:
   percentiles-histogram:
    http.server.requests: true
  tags:
   application: ${spring.application.name}
```

Since edge service is protected by Spring Security, for simplicity, we exclude Actuator from protected endpoints by modifying the **`springSecurityFilterChain`** method of the **`SecurityConfig`** class:

```
@Bean
SecurityWebFilterChain springSecurityFilterChain(
        ServerHttpSecurity http,
        ReactiveClientRegistrationRepository clientRegistrationReposito-
ry) {

    return http
            .authorizeExchange(exchange -> exchange
                    .pathMatchers("/", "/*.css", "/*.js", "/favicon.ico",
                                        "/actuator/**").permitAll()
                    .anyExchange().authenticated())
            .oauth2Login(Customizer.withDefaults())
            .logout(logout -> logout
                    .logoutSuccessHandler(
                          oidcLogoutSuccessHandler(clientRegistrationReposi
tory)))
            .csrf(ServerHttpSecurity.CsrfSpec::disable)
            .build();
```

In a real production contest, Actuator endpoints should also be protected if the microservice is publicly exposed.

In order to know the health status of the edge service, invoke the health endpoint, as shown below:

```
http :8090/actuator/health
```

In order to find out the list of all available metrics, invoke the endpoint, as shown below:

```
http :8090/actuator/metrics
```

In order to know the value of a specific metric, you can add to the path of the metrics endpoint, the name of the metric. For example, to know the memory used by the application, we invoke the following endpoint:

```
http :8090/actuator/metrics/jvm.memory.used
```

It returns a response like the one shown below:

```
{
  "availableTags": [
    {
```

```
      "tag": "area",
      "values": [
        "heap",
        "nonheap"
      ]
    },
    {
      "tag": "application",
      "values": [
        "edge-service"
      ]
    },
    {
      "tag": "id",
      "values": [
        "G1 Survivor Space",
        "Compressed Class Space",
        "CodeCache",
        "G1 Old Gen",
        "Metaspace",
        "G1 Eden Space"
      ]
    }
  ],
  "baseUnit": "bytes",
  "description": "The amount of used memory",
  "measurements": [
    {
      "statistic": "VALUE",
      "value": 187615840.0
    }
  ],
  "name": "jvm.memory.used"
}
```

In the section *Monitoring with Grafana*, we will see how to monitor metrics through a simple dashboard. Add the configurations seen for the edge service also in the catalog service.

# Tracing with Micrometer, OpenTelemetry, and Grafana Tempo

Distributed tracing allows a request to be tracked as it traverses several microservices, providing a clear and detailed view of the path and time taken in each service.

Distributed tracing allows logs to be grouped according to a unique identifier, the **TraceID** which represents the entire end-to-end request or its lifecycle. In addition, for each component involved in the request, another identifier, the **SpanID**, is added, which allows the identification of individual service calls within the same request identified by the TraceID. These identifiers are automatically propagated between microservices via HTTP headers.

Spring Boot 3 enhances support for distributed tracing with the **Micrometer Tracing** project, which replaces Spring Cloud Sleuth. Much like Micrometer's API for metrics, Micrometer Tracing provides a facade for using more popular tracing libraries. The unification of metrics and tracing through Micrometer simplifies monitoring by offering a consistent API and approach for both, making it easier to integrate and manage observability in applications. For our application, we will use **OpenTelemetry**, a vendor-neutral framework for collecting telemetry data (logs, metrics, traces). We will use OpenTelemetry only for distributed tracing. Micrometer Tracing provides a bridge library to use OpenTelemetry, so we will not use the OTEL collector (**https://opentelemetry.io/docs/collector**). We use **Grafana Tempo** as the backend tracing tool, which allows Grafana to query traces. In order to start Tempo locally, add the following piece of code to the **`docker-compose.yml`** file:

```
tempo:
 image: grafana/tempo:2.3.1
 container_name: "easyshop-tempo"
 extra_hosts: ['host.docker.internal:host-gateway']
 command: [ "-config.file=/etc/tempo.yaml" ]
 volumes:
  - ./tempo/tempo-local.yaml:/etc/tempo.yaml:ro
  - ./tempo-data:/tmp/tempo
 ports:
  - "9411:9411" # zipkin
 depends_on:
  - loki
```

Again, you will find the Tempo configuration file in the **infra/docker/tempo** folder. To use Micrometer Tracing with OpenTelemetry in edge service, we add Micrometer's bridge dependency dedicated to OpenTelemetry, as shown below:

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-tracing-bridge-otel</artifactId>
</dependency>
```

In addition, we will have the application send traces in OpenZipkin format, using OpenTelemetry's exporter dependency for Zipkin, as shown below:

```
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-exporter-zipkin</artifactId>
</dependency>
```

By importing this dependency, the edge service will send traces by invoking the Tempo endpoint **http://localhost:9411/api/v2/spans**. Let us move on to the configuration part.

In the **application.yml** file, we disable the export of OpenTelemetry metrics, as we use the latter only for tracing, as demonstrated below:

```
management:
 otlp:
  metrics:
   export:
    enabled: false
```

In addition, we enable tracing and, for simplicity, send all traces to the Zipkin endpoint, setting the sampling probability to 1.0. The complete configuration of the Spring Boot Actuator is as follows:

```
management:
 otlp:
  metrics:
   export:
    enabled: false
 endpoints:
  web:
   exposure:
```

```
    include: '*'
 endpoint:
  health:
   show-details: always
   show-components: always
 tracing:
  enabled: true
  sampling:
   probability: 1.0
 metrics:
  distribution:
   percentiles-histogram:
    http.server.requests: true
  tags:
   application: ${spring.application.name}
```

**Note: By default, the probability of the trace being sampled is set to 0.1 (the range is 0.0-1.0). Enabling sampling at 1.0 in a real production system may result in excessive costs.**

Since the edge service uses a reactive context, we need to enable automatic context propagation. With Spring Boot 3, which uses Project Reactor 3, this is very simple; just add the **`Hooks.enableAutomaticContextPropagation()`** line in **`EdgeServiceApplication`**:

```
@SpringBootApplication
public class EdgeServiceApplication {

    public static void main(String[] args) {
       SpringApplication.run(EdgeServiceApplication.class, args);
        Hooks.enableAutomaticContextPropagation();
    }
}
```

In earlier versions of Spring Boot, context propagation required much more code to be written.

**Note: What is context propagation? The TraceID is added into the logs via the MDC (Mapped Diagnostic Context). MDC is a mechanism provided by logging frameworks like Logback and Log4j, which enriches logs by storing key-value pairs (e.g., TraceID or SpanID) in a thread-local map. In traditional, non-reactive applications, MDC is widely used to correlate logs across the lifecycle of a request since the entire processing flow typically occurs on the same thread. However, in reactive programming, where a single HTTP request may span multiple threads, the use of MDC becomes less effective because thread-local storage does not propagate across threads. To address this, context propagation mechanisms are introduced to ensure that the reactive context is linked with the ThreadLocal objects. This linkage allows the key-value pairs in MDC to be available across different threads transparently, maintaining the correlation of logs even in highly concurrent reactive systems.**

We have finished the tracking configuration in the edge service. Bring the same configurations back into the catalog service. All that remains is to visualize the logs, metrics, and traces on Grafana dashboards.

# Monitoring with Grafana

Grafana is an open-source platform for data visualization and analysis that enables the creation of interactive and customized dashboards. It supports a variety of data sources, including Prometheus, Elasticsearch, AWS CloudWatch, Google Cloud Monitoring, Azure Monitor, and many others. This capability allows data from multiple systems to be integrated into a single visualization platform. It also supports the creation of alerts on specific metrics. Users can set thresholds and receive notifications via email, Slack, or other communication channels when metrics exceed defined threshold values. In our case, Grafana is configured to integrate seamlessly with Loki, Tempo, and Prometheus to display logs, traces, and metrics, respectively. The integration is achieved by defining the data sources in a configuration file, **`datasource.yml`**, as shown below:

```
apiVersion: 1
datasources:
  - name: Prometheus
    type: prometheus
    access: proxy
    url: http://host.docker.internal:9095
    editable: false
    jsonData:
      httpMethod: POST
      exemplarTraceIdDestinations:
        - name: trace_id
```

```
    datasourceUid: tempo
- name: Tempo
  type: tempo
  access: proxy
  orgId: 1
  url: http://tempo:3200
  basicAuth: false
  isDefault: true
  version: 1
  editable: false
  apiVersion: 1
  uid: tempo
  jsonData:
    httpMethod: GET
    tracesToLogs:
      datasourceUid: 'loki'
- name: Loki
  type: loki
  uid: loki
  access: proxy
  orgId: 1
  url: http://loki:3100
  basicAuth: false
  isDefault: false
  version: 1
  editable: false
  apiVersion: 1
  jsonData:
    derivedFields:
      -  datasourceUid: tempo
        matcherRegex: \[.+,(.+?),
        name: TraceID
        url: $${__value.raw}
```

In the **docker-compose.yml** file, we add the Grafana container to run it locally via Docker, as shown below:

```
grafana:
 image: grafana/grafana:10.2.4
 container_name: "easyshop-grafana"
 extra_hosts: ['host.docker.internal:host-gateway']
 volumes:
  - ./grafana/provisioning/datasources:/etc/grafana/provisioning/
datasources:ro
  - ./grafana/provisioning/dashboards:/etc/grafana/provisioning/
dashboards:ro
 environment:
  - GF_AUTH_ANONYMOUS_ENABLED=true
  - GF_AUTH_ANONYMOUS_ORG_ROLE=Admin
  - GF_AUTH_DISABLE_LOGIN_FORM=true
 ports:
  - "3000:3000"
 depends_on:
  - loki
  - prometheus
```

You can find the datasource and dashboard configuration files in the **infra/docker/graphana/provisioning** folder.

We start the containers by running, from the **infra/docker** path, the command: **docker compose up -d**.

We change the visibility of the HTTP logs in our applications from TRACE to INFO so that the consultation of the logs is not too confusing after adding Spring Boot Actuator. In edge service, the log configuration becomes:

```
logging:
 level:
  org.springframework.web: INFO
  io.github.resilience4j: DEBUG
  org.springframework.cloud.gateway: DEBUG
```

In catalog service, on the other hand, the log configuration becomes:

```
logging.level.org.springframework.web=INFO
```

For convenience, we add a log-in catalog service, in the **`findProductByCode`** method of the **`ProductController`** class, which prints the email of the user who invoked the eponymous API, as shown below:

```
@Override
public Mono<ResponseEntity<ProductResponse>> findProductByCode(String productCode,
                                        ServerWebExchange exchange) {
    return ReactiveSecurityContextHolder.getContext()
            .doOnNext(securityContext -> {
                var principal = (Jwt) securityContext
                        .getAuthentication()
                        .getPrincipal();
                log.info("Called findProductByCode with product-
Code {} from {}",
                        productCode, principal.getClaimAsString("email"));
            })
            .flatMap(securityContext ->
                    productService.findProductByCode(productCode))
            .map(ResponseEntity::ok)
            .doOnSuccess(productResponseResponseEntity ->
                    log.info("Terminated findProductByCode"));
}
```

We start edge service and catalog service (remember to bring back the configurations made so far in catalog service as well).

From an incognito tab in the browser, we invoke the URL: **http://localhost:8090/products/LAP-ABCD**. As we saw in the previous chapter, we will be redirected to the Keycloak login page, since we have not yet authenticated ourselves. We enter our username and password (mrossi/mrossi). We will display in response the JSON showing the product information with LAP-ABCD code. Let us look at the edge service logs. We can see that these have been enriched with the TraceID and SpanID, as shown in *Figure 8.2*:

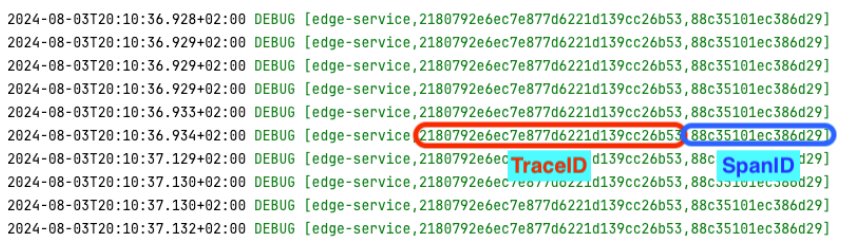



*Figure 8.2*

Look at the catalog service logs as well. You will see that they have been enriched with TraceID and SpanID. Also, since the logs are part of the same HTTP request, the TraceID printed in the catalog service is the same as that of the edge service.

Let us look at Grafana's dashboards. From the browser, we go to the Grafana homepage, at the URL **http://localhost:3000**. We click on the hamburger menu at the top left and select **Dashboards**, as shown in *Figure 8.3*:

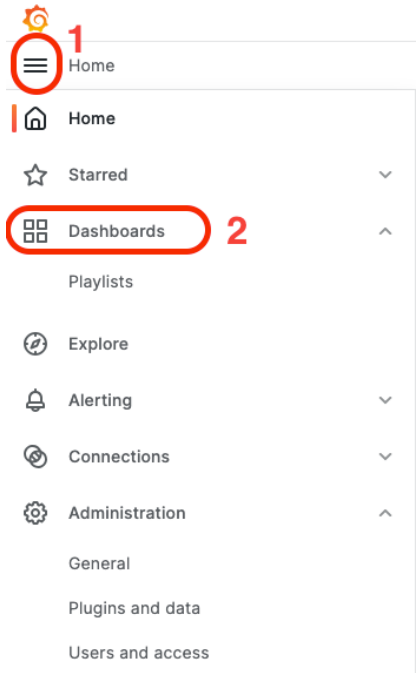



***Figura 8:3**: Grafana's hamburger menu that allows you to select dashboards*

By clicking on *JVM Dashboard*, we will see metrics related to the selected microservice, in this case, catalog service, as shown in *Figure 8.4*. These metrics include resource consumption (e.g., heap and non-heap memory usage), throughput (e.g., HTTP requests per second), and latency (e.g., request duration). Additionally, setting up alerts based on these metrics is essential for proactive monitoring. For example, you can configure alerts to notify operators if the error rate exceeds a certain threshold, if latency surpasses acceptable limits (e.g., p90 or p99 values), or if resource consumption like memory usage nears its maximum capacity. Such alerts ensure timely intervention and maintain system reliability.

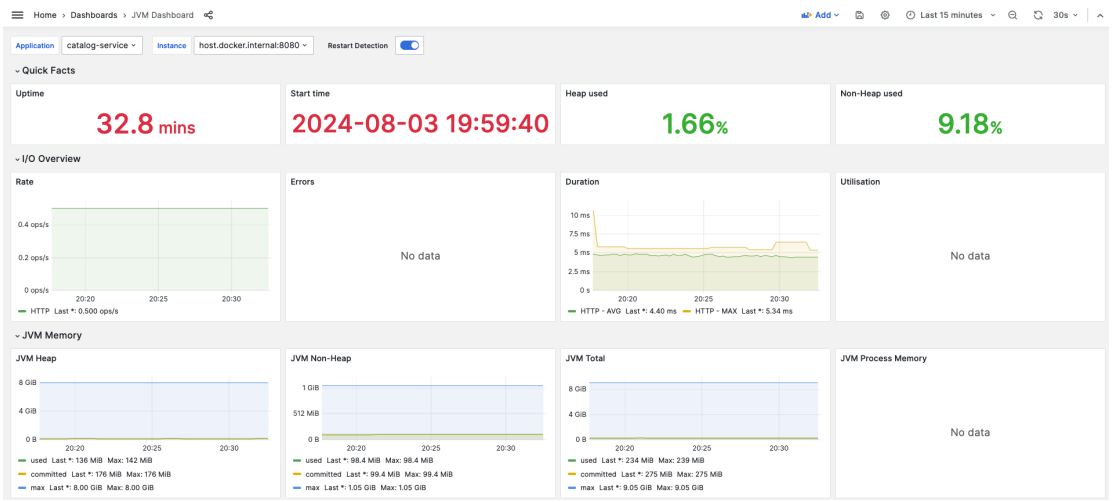



***Figure 8.4:*** *The visualization of Prometheus metrics via Grafana dashboards*

Instead, the **Logs, Traces, Metrics** dashboard shows an overview of logs, traces, and latency, as shown in *Figure 8.5*:

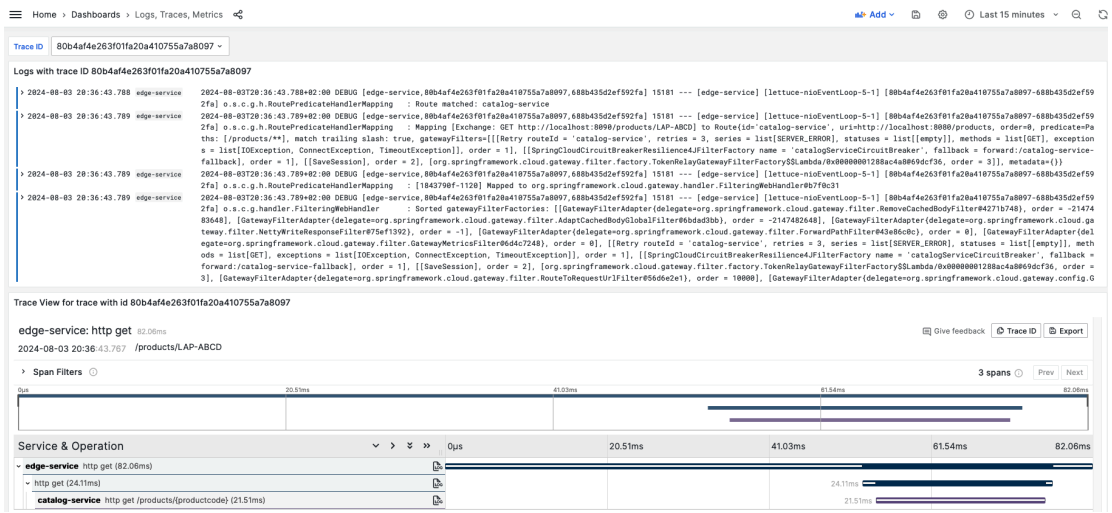



***Figure 8.5:*** *The visualization of the "Logs, Traces, Metrics" dashboard on Grafana*

In the **Explore** panel, found in the hamburger menu, you can view log details (possibly filtered by microservice, level, word contained in the message, and so on), metrics details, and tracing details by selecting the Loki, Prometheus, and Tempo datasources, respectively.

# Conclusion

In this chapter, we have seen how to observe and monitor Spring Boot applications with Spring Boot Actuator and Micrometer. Observability and monitoring become critical in a microservice architecture, where a single HTTP request involves multiple services. With the release of Spring Boot 3, metrics and trace management have been unified by using the Micrometer API for both. In this chapter, we used Prometheus as the metrics store, Loki for the log store, and OpenTelemetry with Tempo to manage traces. We only added configurations, no lines of code needed to be added (net of a single line to enable context propagation in WebFlux). Since the Micrometer provides facades to be used with different tools, we could veer to other tools completely transparently. However, Micrometer also provides APIs for observing specific code points, although we have not discussed them in this chapter. Our applications are ready to be deployed in a production environment. In the next chapter, *Deploying on Kubernetes with Kind*, we will look at how to release the microservices created so far in a containerized context.

# Points to remember

- In a microservice context, distributed tracing is critical. It is strongly discouraged to release an application into production by not managing this aspect.
- In Spring Boot 3, both metrics and tracing are handled by the Micrometer API.
- The Spring Cloud Sleuth Project, which focuses on tracing, has been replaced in Spring Boot 3 by Micrometer Tracing, also developed by the Spring team.

# Exercises

1. Create an alarm in Grafana that will go off if an ERROR level log is present.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

**https://discord.bpbonline.com**



# Chapter 9

# Deploying Applications on Kubernetes with Kind

## Introduction

In the previous chapters, we saw how easy it is to create synchronous and asynchronous APIs with Spring Boot 3. In addition to learning how to write them as efficiently as possible, we also saw in the last part how to shield REST APIs using the edge service microservice, which acts as an API Gateway. We also saw how to secure APIs using Spring Security with the OIDC and OAuth2 standards. Finally, in the last part, we saw how to monitor our systems using Spring Boot Actuator. You now have all the means at your disposal to release into production a system with well-written, secure, and monitorable APIs. However, there are several strategies for releasing to production, however, applications are released as containers. We have already used Docker to run tools such as databases and message brokers locally. In this bonus chapter, we will see how Spring Boot supports out-of-box creation of **Open Container Initiative** (**OCI**) images. We will also see how to release edge service and catalog service microservices on Docker Compose. We will do a theoretical introduction to Kubernetes and then release edge service and catalog service, along with all the other infrastructure tools, on Kubernetes, using *Kind*, which allows you to run a cluster locally.

## Structure

In this chapter, we will discuss the following topics:

- Spring Boot with Docker

- Introduction to Kubernetes
- Deploy your API locally with Kind

# Objectives

The goal of this chapter is not to learn about Docker and Kubernetes technologies in detail, although the initial part of the chapter introduces these tools in a theoretical way. Instead, the goal is to learn what tools Spring provides to make the most of container technology. By the end of this chapter, you will learn how to release Spring Boot applications as containers, either on Docker Compose or Kubernetes, taking full advantage of Spring's features.

# Spring Boot with Docker

With the development of container technology, it was necessary for the most widely used frameworks, such as Spring, to allow OCI images to be created with minimal effort. Spring Boot achieves this by allowing developers to generate images without writing Dockerfiles. This can be done by running the following command in the project root:

```
./mvnw spring-boot:build-image # Maven command
./gradlew bootBuildImage # Grandle command
```

In this section, we will make an introduction to Docker and container technology in general, after which, we will see how to deploy a Spring Boot application as a container.

However, because our microservices will no longer run as standalone applications but as containers, first on Docker Compose and then later a Kubernetes cluster, HTTP requests may take longer to conclude, introducing possible timeouts, particularly on edge service. You can set a timeout of HTTP requests using Resilience4J, by editing the **`application.yml`** file of edge service, and adding a time limiter, for example of five seconds, as shown below:

```
resilience4j:
 circuitbreaker:
  ...
 timelimiter:
  configs:
   default:
    timeoutDuration: 5s
```

# Introduction to container technology and Docker

Before discussing Docker, it is necessary to know what a container is. A container is a

environment. For example, due to this technology, it is possible to run, on the same host, containerized Java applications using different versions of the JDK without having compatibility problems and without performing any configuration. We could define containers as isolated processes: the isolation is achieved due to Linux's cgroup and namespace features. However, although containers are based on Linux features, it is also possible to run them in a macOS and Windows environment thanks to tools, such as *Docker Desktop* or *Podman Desktop*, which virtualize Linux transparently to the user.

Compared to virtual machines, containers are lightweight processes because they share the kernel of the host machine. *Figure 9.1* shows a comparison between containers and machines:

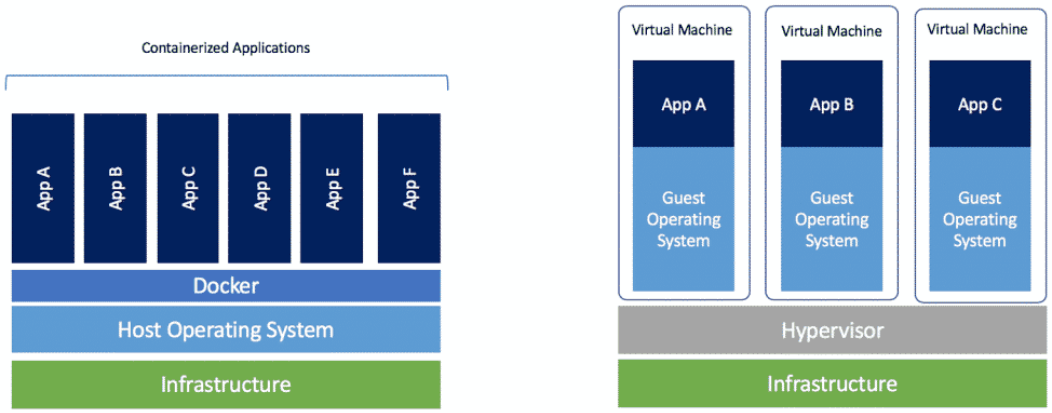



***Figure 9.1**: Comparison between containers and virtual machines*

Virtual machines make it possible to virtualize an entire operating system, with its own Kernel. They work thanks to a hardware or software component called a Hypervisor, which is sandwiched between the host and guest operating systems. Virtual machines, in addition to offering software virtualization, as they boot an entire operating system, also virtualize hardware as it is possible to virtualize the amount of RAM and CPU to be dedicated, networks, and more, thus providing greater isolation and flexibility.

Containers, on the other hand, are started using a Container Runtime, such as Docker. Containers share the same Kernel as the host. This feature makes them lightweight, both in terms of storage (we are in the megabyte range versus the gigabyte range of virtual machines) and in terms of computational resources. Containers *virtualize* the application, Runtime environment, dependencies, and the libraries needed to run it, a feature that makes them more scalable than virtual machines, as VMs virtualize an entire operating system. However, in a real-world scenario, virtual machines and containers are used together and not in substitution for each other. Containers are typically deployed on virtual machines, which serve as their foundational infrastructure.

You might wonder when and how container technology developed. This technology is

architecture came the need to better isolate (micro) applications. Container technology was a perfect fit for this use case. Thus, tools such as Docker emerged to facilitate application deployment via containers.

Docker is a technology for creating and managing containers. As a Container Runtime, Docker not only allows containers to be run from OCI images, but also takes care of other aspects such as networking and persistence management. It has a client-server architecture. The server part is called the Docker Engine, and it is where the dockerd daemon is running that allows container lifecycle management, including creating, running, stopping, and deleting containers. The client part is managed by the Docker CLI which allows command line instruction to interact with the Docker Engine through commands like **`docker build`**, **`docker run`**, and **`docker ps`**. However, there are also graphical clients to manage containers visually. When you install Docker, both the client and server parts are installed. However, nothing prohibits connecting the Docker CLI to a remote Docker Engine. *Figure 9.2* shows an overview of Docker's architecture:

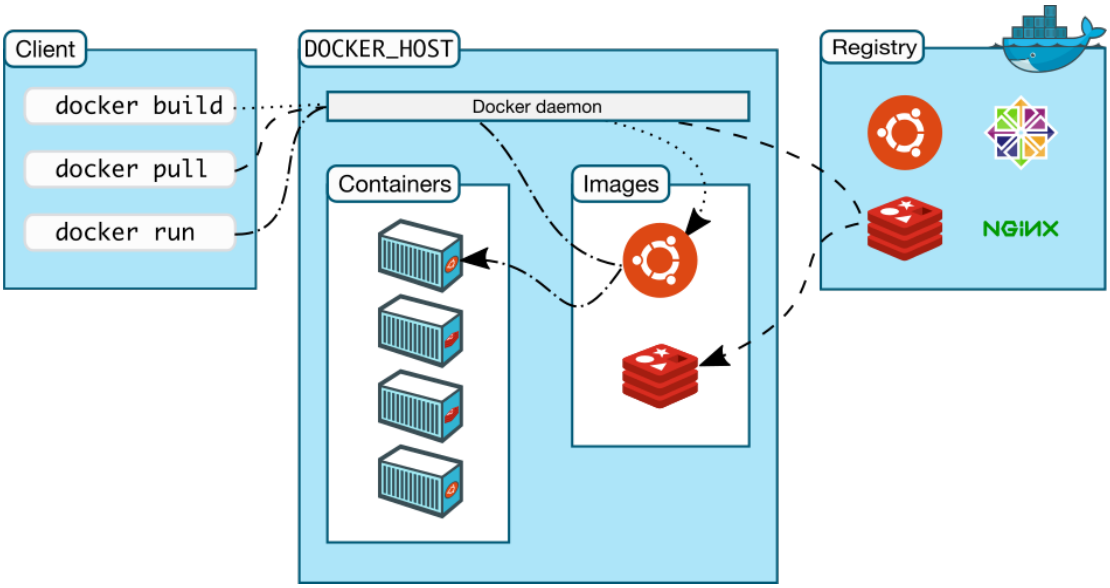



***Figure 9.2**: An overview of Docker's client-server architecture*

Containers are created from Docker images or from OCI-compliant images. Images are templates for creating containers. You can create images from a file called Dockerfile, which contains useful commands for building the image. You can also download images from public or private registries (the Docker Hub, **https://hub.docker.com**, is the most popular public registry).

Now that we have seen what container technology and Docker is, we can see how to create containers with Spring Boot.

# Containerizing a Spring Boot application

Normally, to containerize an application, it would be necessary to write a Dockerfile. A Dockerfile is a text file containing commands to assemble an image. For example, to create an image of a Spring Boot application, we might write the following Dockerfile:

```
FROM openjdk:17-jdk-alpine
RUN addgroup -S spring && adduser -S spring -G spring
USER spring:spring
ARG DEPENDENCY=target/dependency
COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY ${DEPENDENCY}/META-INF /app/META-INF
COPY ${DEPENDENCY}/BOOT-INF/classes /app
ENTRYPOINT ["java","-cp","app:app/lib/*","hello.Application"]
```

As shown by the example above, a source image, in this case that of OpenJDK, is always used to create an image. However, writing the Dockerfile involves several challenges that are not easy to address, such as creating efficient and secure images. Another problem with the Dockerfile is the reproducibility of the images. If we create two images for the same application, with the same source code and the same Dockerfile, at two different instants, the two images will have different sha (digests).

Spring Boot uses the **Cloud Native Buildpacks** (**CNBs**) project to create OCI images. CNBs is an open-source project that includes a set of buildpacks that provide a standard for turning application source code into a production-ready OCI image, without the need to write a Dockerfile.

CNBs are particularly useful when you want a standardized and automated process for building images without manually maintaining Dockerfiles, ensuring consistency and reproducibility across builds. They are well-suited for applications adhering to modern development workflows or requiring integration with CI/CD pipelines.

However, Dockerfiles offer greater flexibility and control over the image creation process, making them a better choice for scenarios where custom configurations, specific base images, or unique build steps are required. Choosing between CNBs and Dockerfiles depends on the level of customization and automation needed in your project.

Specifically, Spring Boot uses Paketo Buildpacks, an implementation of the CNBs project, specific to Java (and other) applications.

The buildpack components that help create the application image are mentioned below:

- **Builder:** It is an image that contains a set of buildpacks that provide dependencies for the app.

- **Buildpacks:** They examine the source code of the application, identify dependencies, and produce an OCI image of the app with its dependencies.
- **Stack:** It provides the OS layer for the image and constitutes the base image of the builder.

In order to create an OCI image with Spring Boot, you need to import the Spring Boot Maven plugin, as shown below:

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

Having created all our microservices from Spring Initializr, they already import the plugin.

We create the edge service OCI image. From the root path of the microservice, we run the following command:

```
./mvnw clean spring-boot:build-image
```

Let us look at the first part of the output of the executed command:

```
...
[INFO]  > Pulled builder image 'paketobuildpacks/builder-jammy-base
...
[INFO]     [creator]     ===> DETECTING
...
[INFO]     [creator]     6 of 26 buildpacks participating
[INFO]   [creator]   paketo-buildpacks/ca-certificates  3.8.4
[INFO]   [creator]   paketo-buildpacks/bellsoft-liberica 10.8.2
[INFO]   [creator]   paketo-buildpacks/syft       1.47.1
[INFO]   [creator]   paketo-buildpacks/executable-jar  6.11.0
[INFO]   [creator]   paketo-buildpacks/dist-zip     5.8.2
[INFO]   [creator]   paketo-buildpacks/spring-boot    5.31.0
```

From the log above, we can learn about the builder that is used. The builder above uses the Ubuntu Jammy Jellyfish build with buildpacks for Java and other programming languages. After analyzing the application, only six buildpacks are used. Here is a breakdown of their roles:

- **paketo-buildpacks/ca-certificates**: Installs SSL/TLS certificates to enable secure communication.
- **paketo-buildpacks/bellsoft-liberica**: Provides the Liberica JDK, which is used to run Java applications.

- **paketo-buildpacks/syft**: Scans the application for dependencies to generate **Software Bill of Materials** (**SBoM**) information.
- **paketo-buildpacks/executable-jar**: Prepares the application to run as an executable JAR file.
- **paketo-buildpacks/dist-zip**: Supports applications packaged as a distribution ZIP file.
- **paketo-buildpacks/spring-boot**: Provides optimizations and support specifically for Spring Boot applications.

As you can see, by default, *Paketo* uses the *Liberica JDK* (**`paketo-buildpacks/bellsoft-liberica`**).

Let us analyze the next logs:

```
...
[INFO]     [creator]       $BP_JVM_JLINK_ENABLED        false
[INFO]     [creator]       $BP_JVM_TYPE                 JRE
[INFO]     [creator]       $BP_JVM_VERSION              17
[INFO]     [creator]       Launch Configuration:
[INFO]     [creator]       $BPL_DEBUG_ENABLED           false
[INFO]     [creator]       $BPL_DEBUG_PORT              8000
[INFO]     [creator]       $BPL_JVM_THREAD_COUNT        250
[INFO]     [creator]       $JAVA_TOOL_OPTIONS
[INFO]     [creator]       Using Java version 21 extracted from MANIFEST.MF
[INFO]     [creator]       BellSoft Liberica JRE 21.0.4: Contribut-
ing to layer
[INFO]     [creator]       Downloading from https://github.com/bell-sw/Li-
berica/releases/download/21.0.4+9/bellsoft-jre21.0.4+9-linux-amd64.tar.gz
...
[INFO] Successfully built image 'docker.io/library/edge-service:0.0.1-SNAP-
SHOT'
```

From the log, the values of some properties that contribute to image construction are shown.

**Note: Although the BP_JVM_VERSION property is valued at 17, the builder, upon analyzing the manifest, understands that the correct version of the JDK to use is 21. You can customize the name of the builder to use, the name of the final image, and more from the command line.**

For example, to change the name of the image, you can use the following command:

```
./mvnw spring-boot:build-image \
-Dspring-boot.build-image.imageName=springio/gs-spring-boot-docker
```

You can find the exhaustive list of properties that customize the image, on the official documentation here: **https://docs.spring.io/spring-boot/maven-plugin/build-image.html#build-image.customization**

You can customize the Spring Boot Maven plugin to set environment variables that are useful for building the image. The following code snippet demonstrates how to configure the plugin to set the Java version to 21 during the image build process:

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <image>
      <env>
        <BP_JVM_VERSION>21</BP_JVM_VERSION>
      </env>
    </image>
  </configuration>
</plugin>
```

Now that we have seen how to create container images with Spring Boot, we will leave it to you to create the microservice catalog service image.

# Running Spring Boot containers with Docker Compose

Check that the edge service and catalog service images have been created correctly, using the **`docker images`** command. You should find in the list of images you have locally, as well as those of the two microservices. We will create the two containers with Docker Compose, using the same file we have used so far. All containers in the same Docker Compose file share the same network. Containers on the same network can communicate with each other and are isolated from containers belonging to other networks. In fact, Docker automatically creates on each container, a DNS record that associates the names of containers in the same network, with their IP. Then, edge service can invoke the catalog service using the URL: **http://catalog-service:8080**.

In traditional applications, a different configuration file was created for each environment. For example, we would have an

configurations, **`application-uat.properties`** for those in the UAT environment, and so on. However, Cloud Native applications follow a different approach. Configurations must be made explicit through environment variables, not through different files. For each environment, the values of the environment variables change.

Spring Boot fully espouses this approach. In fact, each property set in the **`application.properties`** file can be overwritten with the associated environment variable, using a naming-convention. For example, the property defined in the previous chapter, **`loki.endpoint`**, can be overridden by the environment variable **`LOKI_ENDPOINT`**. You can also define custom environment variables that can be used in the **`application.properties`** file. For example, in the edge-service **`application.yml`** file, we modify the property inherent in the catalog service URL, as shown below:

```
routes:
 - id: catalog-service
   uri: ${CATALOG_SERVICE_URL:http://localhost:8080}/products
```

The above piece of code assigns to the URL of the route with id "**`catalog-service`**", the value of the environment variable **`CATALOG_SERVICE`**, if it presents, otherwise it assigns it the default value, the one after the colon (:), i.e. **http://localhost:8080**.

Leveraging the concepts above, we create Docker Compose services for edge service and catalog service, adding them in the **`infra/docker/docker-compose.yml`** file, as shown below:

```
edge-service:
 image: edge-service:0.0.1-SNAPSHOT
 container_name: edge-service
 extra_hosts: ['easyshop-keycloak.com:host-gateway']
 ports:
  - "8090:8090"
 environment:
  CATALOG_SERVICE_URL: http://catalog-service:8080
  SPRING_SECURITY_OAUTH2_CLIENT_PROVIDER_KEYCLOAK_ISSUER_URI:
   http://host.docker.internal:8091/realms/Easyshop
  SPRING_DATA_REDIS_HOST: easyshop-redis
  MANAGEMENT_ZIPKIN_TRACING_ENDPOINT: http://tempo:9411/api/v2/spans
  LOKI_ENDPOINT: http://easyshop-loki:3100
 depends_on:
  - pg
  - keycloak
```

```
  - loki
  - prometheus
  - redis
catalog-service:
 image: catalog-service:0.0.1-SNAPSHOT
 container_name: catalog-service
 extra_hosts: ['easyshop-keycloak.com:host-gateway']
 user: root
 ports:
  - "8080:8080"
 environment:
  SPRING_R2DBC_URL:
   r2dbc:postgresql://easyshop-postgres:5432/easyshopdb_catalog
  SPRING_R2DBC_USERNAME: user
  SPRING_R2DBC_PASSWORD: password
  SPRING_SECURITY_OAUTH2_RESOURCESERVER_JWT_ISSUER_URI:
   http://host.docker.internal:8091/realms/Easyshop
  SPRING_DATA_REDIS_HOST: easyshop-redis
  MANAGEMENT_ZIPKIN_TRACING_ENDPOINT: http://tempo:9411/api/v2/spans
  LOKI_ENDPOINT: http://easyshop-loki:3100
 depends_on:
  - pg
  - keycloak
  - loki
  - prometheus
```

Note that because we use the OIDC flow, to make the URL redirects work from our computer's browser, edge servie, and catalog service do not refer to the Keycloak container via internal DNS, but via the DNS **`host.docker.internal`**, which can be used inside Docker containers to automatically relink the host IP. In order to be able to use it outside of containers as well, add to your computer's host file the record, as demonstrated below:

```
127.0.0.1 host.docker.internal
```

Now, we are ready to start the containers. First, we start the containers of the tools used by edge service and catalog service, using the following command, from the path **`infra/docker`**:

```
docker compose up -d pg keycloak redis tempo loki prometheus grafana
```

Wait a few minutes so that Keycloak has completed all configurations. Then, we also start edge service and catalog service with the following command:

```
docker compose up -d edge-service catalog-service
```

Open an incognito browser tab and type the usual URL: **http://localhost:8090/products/LAP-ABCD**. You will be redirected correctly to the Keycloak login page, at the URL host-**`docker.internal`**. After that, enter the credentials **`mrossi/mrossi`**. You will be correctly redirected to the catalog service response containing the details of the requested product.

In this section, we saw how to create container images with Spring Boot and how to run edge service and catalog service containers with Docker Compose. Due to Spring Boot, writing Dockerfiles to create the images was unnecessary. However, you will rarely see Docker Compose used in a production system. This is because Docker Compose is primarily designed for local development and testing environments, lacking key features required for production workloads. For example, it does not provide advanced scheduling, scaling, or failover mechanisms. To release containers in production, you should use a container orchestrator such as Kubernetes, which addresses these limitations by providing robust capabilities for scaling, high availability, and fault tolerance. To release containers in production, you should use a container orchestrator such as Kubernetes.

# Introduction to Kubernetes

Kubernetes, often abbreviated as K8s, is an open-source container orchestration platform that automates the deployment, management, and scalability of containerized applications. Originally developed by *Google* and later donated to the **Cloud Native Computing Foundation** (**CNCF**), Kubernetes has become the standard for managing modern, container-based applications in production environments.

The development of microservices architecture has caused container technology to become increasingly popular. However, while they are lightweight and portable entities, managing hundreds or thousands of containers in production environments can become complex and error-prone.

Kubernetes addresses these challenges by providing a robust, automated infrastructure for large-scale container control and management. In particular, Kubernetes provides the following features:

- **Orchestration:** Kubernetes automates container deployment and management, reducing operational load and improving application reliability.
- **Dynamic scalability:** It allows applications to scale automatically based on demand. Kubernetes continuously monitors resources and can add or remove containers based on application needs.
- **Self-healing:** Kubernetes automatically detects container failures and restarts them if necessary, ensuring that the application remains operational even in the event of unexpected failures.

- **Load management:** Automatically distributes network traffic among containers, balancing the load to optimize performance and prevent overloading on individual components.
- **Rolling updates:** Supports rolling updates of applications, allowing new versions of software to be deployed in a safe and controlled manner, without having downtime.
- **Integration with Cloud ecosystems:** Kubernetes natively integrates with leading cloud providers, such as *AWS*, *Google Cloud*, and *Microsoft Azure*, facilitating the deployment of applications in hybrid and multi-cloud environments.

Although the architecture of Kubernetes is complex, we can list the main components, as mentioned below:

- **Kubernetes cluster:** A cluster consists of a set of nodes (physical or virtual servers). A cluster consists of at least one *control plane* type node and one or more *worker* type nodes.
- **Worker nodes:** Each worker node in a Kubernetes cluster runs one or more Pods (groups of one or more containers). Worker nodes include components such as **kubelet** (an agent that runs containers), **`kube-proxy`** (networking management), and Container Runtime (e.g., Docker or containerd).
- **Control plane:** The control plane is responsible for managing the Kubernetes cluster and monitoring the Pods. It includes components such as **`kube-apiserver`** (API interface), **`etcd`** (distributed database), **`kube-scheduler`** (container placement management), and **`kube-controller-manager`** (controller for cluster state management). To ensure reliability, more control plane nodes can be added. This redundancy is crucial because if the control plane becomes unavailable, the cluster might not be able to create new Pods, scale up or down, or maintain the desired state. Having multiple control plane nodes allows the cluster to remain functional, as other nodes can take over in case of a failure, minimizing downtime and ensuring consistent operations.
- **Pod:** The Pod is the basic unit of execution in Kubernetes and can contain one or more containers that share the same network and storage space. Pods are ephemeral entities, meaning they can be terminated and replaced automatically. To manage the number or availability of Pods, you can configure the maximum and minimum replicas in a ReplicaSet. Additionally, the **Horizontal Pod Autoscaler** (HPA) can be used for dynamic scaling, adjusting the number of Pods based on resource usage or custom metrics.
- **Service and Ingress/Gateway:** Kubernetes uses Services to expose Pods inside or outside the cluster, ensuring communication between application components. Ingress and Gateway provide a way to manage external access to applications.

*Figure 9.3* shows a diagram summarizing the elements of the Kubernetes architecture:

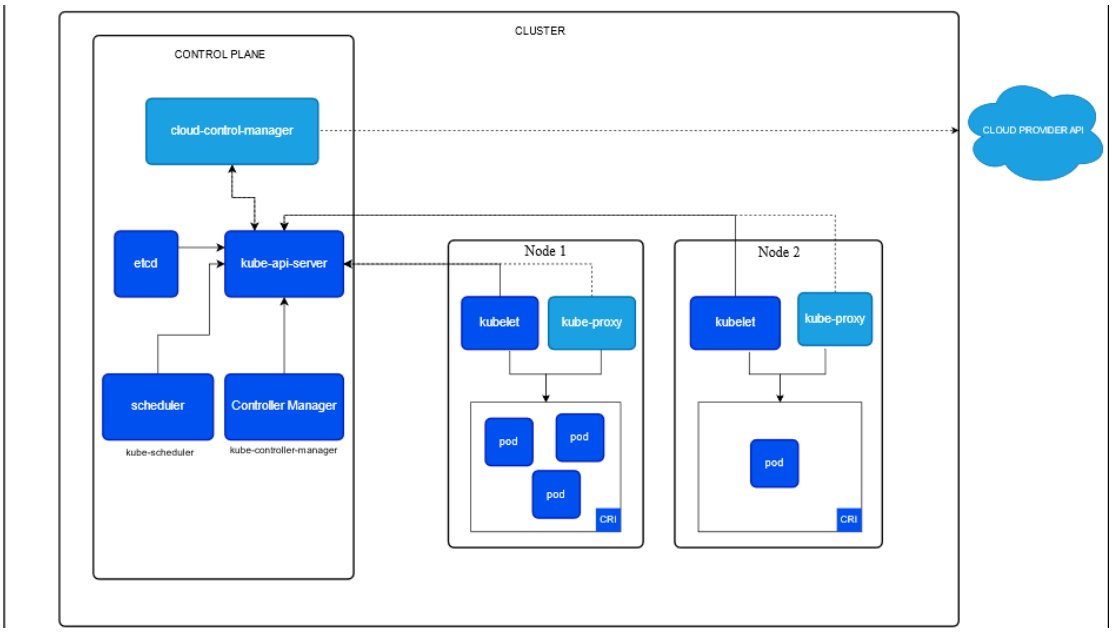



***Figure 9.3**: Kubernetes architecture* [1]

# Deployment and service in Kubernetes

Containers in Kubernetes are wrapped in Pod type objects. Pods can contain one or more containers. In reality, Pods contain only one main container plus any containers that act as a sidecar, such as **Init Containers** and **Sidecar Containers**. Init containers are run before the main container to perform preliminary operations useful for the operation of the main container. Sidecar containers, on the other hand, implement the **sidecar pattern**, which pairs a secondary process or service (the "sidecar") with a primary application to handle complementary tasks. These tasks can include logging, monitoring, proxying, security, or configuration management. The sidecar runs alongside the main application, sharing the same Pod, but remains logically and operationally independent, ensuring modularity and separation of concerns.

Kubernetes objects are deployable declaratively using YAML files. An example of a YAML file that allows the creation of a Pod containing a Nginx container is as follows:

```
apiVersion: v1
kind: Pod
metadata:
 name: nginx
```

1 image taken from **https://kubernetes.io/docs/concepts/architecture**

```
spec:
 containers:
 - name: nginx
  image: nginx:1.14.2
  ports:
  - containerPort: 80
```

However, you will never directly create Pod objects. Rather, you will create resources of type **Deployment**. A resource of type Deployment creates an object of type **ReplicaSet**. In turn, the **`ReplicaSet`** object creates a set of Pods, ensuring that the number of instances running is always the desired number, that is, the number specified in the YAML file. The **`Deployment`** object adds functionality to the ReplicaSet deployment, for example, when the deployment YAML file is updated, such as because you want to update the version of the container image, the **`Deployment`** object creates another ReplicaSet and automatically handles the destruction of the old one. The following is an example of a YAML file for creating a **`Deployment`** object that creates three Nginx Pods:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
 labels:
  app: nginx
spec:
 replicas: 3
 selector:
  matchLabels:
   app: nginx
 template:
  metadata:
   labels:
    app: nginx
  spec:
   containers:
   - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

In the YAML file above, a Deployment is created that creates a RepliceSet, which in turn creates three Nginx Pods. The **`spec.selector`** field defines which Pods the RepliceSet should handle, in this case, all those that match the label "app: nginx." In fact, the Pods created in this file have the label "app: nginx" (**`spec.template.metadata.labels field`**).

Another fundamental object of Kubernetes is the **Service**. Each Pod has its own logical IP. However, Pods are ephemeral entities, they are created and destroyed continuously by Kubernetes to always maintain the desired state. Each time a Pod is created, it is assigned a different IP. Take the example of edge service, which makes requests to catalog service. Edge service cannot invoke catalog service via a Pod IP, due to the ephemeral nature of the Pod. The **`Service`** object allows a set of Pods to be exposed using the selector, as with Deployments with ReplicaSets, via its logical name. An example of a Service that allows exposing service catalogs within the Kubernetes cluster is as follows:

```
apiVersion: v1
kind: Service
metadata:
 name: catalog-service
 labels:
  app: catalog-service
spec:
 type: ClusterIP
 selector:
  app: catalog-service
 ports:
  - protocol: TCP
   port: 8080
   targetPort: 8080
```

This Service is associated with all Pods that have the label "app: catalog-service" and allows them to be exposed within the cluster via the logical catalog-service name (**`metadata.name`** field). The **`Service`** object will take care of routing the requests it receives to a Pod that is *healthy* and *ready* to receive traffic. In the following section, we will see how to handle the healthy and ready states. Services in turn can be exposed on the Internet using Ingress objects or the more modern Gateways. However, we have four types of Services, as mentioned below:

- **ClusterIP**: The default, exposes the Service on a cluster-internal IP.
- **NodePort**: This exposes the service on the IPs of each cluster node, via a static port.
- **LoadBalancer**: This exposes the service externally using a load balancer not directly managed by Kubernetes, which can be provided manually or by the Cloud Provider.

- **ExternalName**: This exposes the service via a DNS name instead of via selector.

**Note: The type field of the Service object is designed as a nested feature, each level is added to the previous one. For example, by creating a Service of type NodePort, a clusterIP is created. However, there is an exception to this nested structure. A Service of type LoadBalancer can be defined by disabling the allocation of load balancer NodePorts.**

In this section, we have learned about the basic elements of Kubernetes, now all that remains is to put these concepts into practice in our local environment.

# Deploy your API locally with Kind

In order to create a Kubernetes cluster locally, we have several tools available that allow us to do so, such as Docker Desktop itself. However, in this book, we have chosen to use Kind (Kubernetes in Docker), a tool that allows you to run a Kubernetes cluster locally using containers that act as cluster nodes. Kind allows you, unlike other tools, to create a multi-node cluster in a customized way, for example, you can set how many worker nodes and how many control plane nodes to create.

## Installation and creation of the cluster

You can install Kind on Linux, Windows, and macOS. On macOS and Windows, you can use installation via Package Manager by running the following commands:

```
brew install kind # For macOS
```

```
winget install Kubernetes.kind #For Windows
```

You can also install Kind via release binaries. On Linux, the command is as follows:

```
# For AMD64 / x86_64
```

```
[ $(uname -m) = x86_64 ] && curl -Lo ./kind https://kind.sigs.k8s.io/dl/
v0.24.0/kind-linux-amd64
```

```
# For ARM64
```

```
[ $(uname -m) = aarch64 ] && curl -Lo ./kind https://kind.sigs.k8s.io/dl/
v0.24.0/kind-linux-arm64
```

```
chmod +x ./kind
```

```
sudo mv ./kind /usr/local/bin/kind
```

On MacOS, the command is as follows:

```
# For Intel Macs
[ $(uname -m) = x86_64 ] && curl -Lo ./kind https://kind.sigs.k8s.io/dl/
v0.24.0/kind-darwin-amd64
# For M1 / ARM Macs
[ $(uname -m) = arm64 ] && curl -Lo ./kind https://kind.sigs.k8s.io/dl/
v0.24.0/kind-darwin-arm64
chmod +x ./kind
mv ./kind /some-dir-in-your-PATH/kind
```

Finally, on Windows, the command to run via PowerShell is as follows:

```
curl.exe -Lo kind-windows-amd64.exe https://kind.sigs.k8s.io/dl/v0.24.0/
kind-windows-amd64

Move-Item .\kind-windows-amd64.exe c:\some-dir-in-your-PATH\kind.exe
```

Remember to add the path to the Kind executable in your operating system's PATH environment variable so that you can run the **`kind`** command from any path. For more details on installing Kind, you can visit the official documentation: **https://kind.sigs.k8s.io/docs/user/quick-start/#installation**.

To verify that Kind has been installed correctly, run the following command:

```
kind --version
```

You should have in response the version of Kind installed.

Kind does not automatically install the command-line tool "kubectl", which allows you to perform operations on the cluster, such as creating resources. This tool is also available for Linux, macOS and Windows. Look at the official documentation to install it: **https://kubernetes.io/docs/tasks/tools/#kubectl**.

In order to create the cluster, we will use the **`kind-multi-node-cluster.yaml`** file that allows you to create one node of the type control plane and two nodes of the type worker. You can find this file in the Git repository of the book, at the **`infra/kind`** path in the **chapter-09** folder. From the command line, go to the **`infra/kind`** path and run the following command that allows you to create the cluster:

```
kind create cluster --config kind-multi-node-cluster.yaml --name easyshop-
cluster
```

You should have output similar to that shown in *Figure 9.4*, certifying that the cluster has been created correctly:

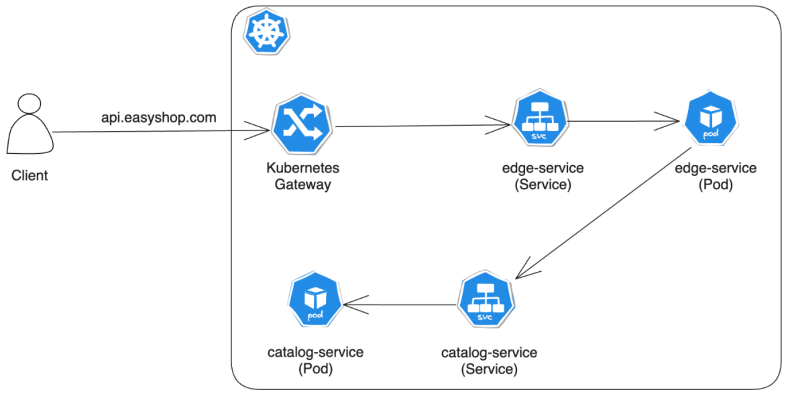



***Figure 9.4**: The output of Kind's command to create a cluster locally*

As suggested by the output above, also run the following command:

```
kubectl cluster-info --context kind-easyshop-cluster
```

Well, now we are ready to interact with the Kubernetes cluster. Let us verify that the nodes have been created correctly, using the following command:

```
kubectl get no
```

You should see the following output:

```
NAME                              STATUS   ROLES           AGE     VERSION
easyshop-cluster-control-plane    Ready    control-plane   7m40s   v1.30.2
easyshop-cluster-worker           Ready    <none>          7m19s   v1.30.2
easyshop-cluster-worker2          Ready    <none>          7m19s   v1.30.2
```

The nodes are all in the *ready* state, so we can create the resources for the proper operation of Easyshop.

**Note: If your computer has difficulty running three cluster nodes, you can lighten the cluster by creating fewer nodes by editing the kind-multi-node-cluster.yaml file.**

In order to delete the newly created cluster, run the following command:

```
kind delete cluster --name easyshop-cluster
```

# Deploy Easyshop to Kubernetes

As mentioned earlier, we will deploy only the edge service and catalog service to Kubernetes, but the process is the same for all other microservices. In addition, we will

also release on Kubernetes the tools that have been running so far with Docker Compose, such as Keycloak and Postgres, however, the release mode of these components is only suitable for a local and not a production environment. *Figure 9.5* shows an overview of the components involved in an HTTP request:

```
kind create cluster --config kind-multi-node-cluster.yaml --name easyshop-cluster
Creating cluster "easyshop-cluster" ...
 ✓ Ensuring node image (kindest/node:v1.30.2) 🖼
 ✓ Preparing nodes 📦 📦 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
 ✓ Joining worker nodes 🚜
Set kubectl context to "kind-easyshop-cluster"
You can now use your cluster with:

kubectl cluster-info --context kind-easyshop-cluster

Have a question, bug, or feature request? Let us know! https://kind.sigs.k8s.io/#community 🙂
```

***Figure 9.5**: Overview of the Kubernetes elements involved*

As shown by the figure above, we will use Kubernetes' Gateway object to externally expose edge service APIs. The Gateway is a recent object that replaces the Ingress object, the goal is the same, which is to externally expose one or more Services, but the Gateway has more advanced features to learn more about the Gateway, visit: **https://kubernetes.io/docs/concepts/services-networking/gateway**.

In order to simulate as much as possible a real case, the Gateway will respond to all requests having the hostname: *api.easyshop.com*. We will then add the following line to the hosts file of our computer:

```
127.0.0.1 api.easyshop.com
```

## Writing deployment and service

Pods can provide information about their internal state through container probes. A **probe** is a diagnostic performed periodically by kubelet on the container. Specifically, there are three types of probes:

- **livenessProbe:** It indicates whether the container is running. If the liveness probe fails, the kubelet will kill the container. The container will then be recreated or not depending on the Pod's restartPolicy.
- **readinessProbe:** It indicates whether the container is ready to respond to requests. If the readiness probe fails, the Service will not turn requests on that Pod.
- **startupProbe**: It indicates whether the containerized application is started. If the `startupProbe`

is defined, the delay (**`initialDelaySeconds`**) of the liveness probe and readiness probe does not start until the startup probe succeeds.

Spring Boot manages out-of-box application state via **`livenessProbe`** and **`readinessProbe`**, with the Spring Boot Actuator module, which we have already used, so we do not need to modify the code. The probes are shown via the **`"/actuator/health/liveness"`** and **`"/actuator/health/readiness"`** endpoints.

That said, we can write the YAML file to create the edge service **`Deployment`** object, as shown below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: edge-service
  labels:
    app: edge-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: edge-service
  template:
    metadata:
      labels:
        app: edge-service
    spec:
      containers:
        - name: edge-service
          image: edge-service:0.0.1-SNAPSHOT
          ports:
            - containerPort: 8090
          env:
            - name: CATALOG_SERVICE_URL
              value: http://catalog-service:8080
            - name: SPRING_SECURITY_OAUTH2_CLIENT_PROVIDER_KEYCLOAK_ISSUER_
URI
              value:
```

```
            - name: SPRING_DATA_REDIS_HOST
              value: easyshop-redis
            - name: MANAGEMENT_ZIPKIN_TRACING_ENDPOINT
              value: http://easyshop-tempo:9411/api/v2/spans
            - name: LOKI_ENDPOINT
              value: http://easyshop-loki:3100
          livenessProbe:
            httpGet:
              path: /actuator/health/liveness
              port: 8090
            initialDelaySeconds: 10
            periodSeconds: 5
          readinessProbe:
            httpGet:
              path: /actuator/health/readiness
              port: 8090
            initialDelaySeconds: 15
      periodSeconds: 15
```

The following points summarize the key components and configurations of the Deployment described in the example:

- The Deployment creates a ReplicaSet that creates a Pod (**`spec.replicas`**).
- The ReplicaSet manages Pods that have the label **`"app: edge-service"`** (**`spec.selector`**).
- The created Pod has the label **`"app: edge-service"`** (**`spec.template.metadata.labels`**).
- HTTP endpoints for liveness probes and readiness probes are defined. The kubelet will invoke the liveness endpoint every five seconds, with an initial delay of ten, while the readiness endpoint will be invoked every 15 seconds, with an initial delay of 15 seconds.

The Deployment file of the catalog service is very similar to that of the edge service, as demonstrated below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: catalog-service
  labels:
    app: catalog-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: catalog-service
  template:
    metadata:
      labels:
        app: catalog-service
    spec:
      containers:
        - name: catalog-service
          image: catalog-service:0.0.1-SNAPSHOT
          ports:
            - containerPort: 8080
          env:
            - name: SPRING_R2DBC_URL
              valueFrom:
                secretKeyRef:
                  name: postgres-secret
                  key: url
            - name: SPRING_R2DBC_USERNAME
              valueFrom:
                secretKeyRef:
                  name: postgres-secret
                  key: username
            - name: SPRING_R2DBC_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: postgres-secret
                  key: password
```

```
        - name: SPRING_SECURITY_OAUTH2_RESOURCESERVER_JWT_ISSUER_URI
          value: http://host.docker.internal:8091/realms/Easyshop
        - name: MANAGEMENT_ZIPKIN_TRACING_ENDPOINT
          value: http://easyshop-tempo:9411/api/v2/spans
        - name: LOKI_ENDPOINT
          value: http://easyshop-loki:3100
      livenessProbe:
        httpGet:
          path: /actuator/health/liveness
          port: 8080
        initialDelaySeconds: 10
        periodSeconds: 5
      readinessProbe:
        httpGet:
          path: /actuator/health/readiness
          port: 8080
        initialDelaySeconds: 15
        periodSeconds: 15
```

The only thing relevant in this file is that environment variables affecting Postgres are not valued directly but are extracted from a Secret called **`postgres-secret`**. A Secret is a Kubernetes resource that allows sensitive properties to be defined via a value key map. If the data were not sensitive, a resource called ConfigMap could be used for the same purpose. Both are useful for decoupling configurations from Deployment.

Both Deployments must then be exposed via Services. For both edge service and catalog service, ClusterIP type Services are sufficient, so they will be exposed within the cluster. The Service of edge service is as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: edge-service
  labels:
    app: edge-service
spec:
  type: ClusterIP
  selector:
```

```
    app: edge-service
  ports:
    - protocol: TCP
      port: 8090
      targetPort: 8090
```

The port field indicates the port exposed by the Service, while the **`targetPort`** field specifies the Pod port to which the Service should route traffic.

The Service of catalog service is similar to the previous one, as shown below:

```
apiVersion: v1
kind: Service
metadata:
  name: catalog-service
  labels:
    app: catalog-service
spec:
  type: ClusterIP
  selector:
    app: catalog-service
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

Then, the edge service will be able to invoke catalog service at the URL **http://catalog-service:8080**.

Finally, as shown in the overview, we expose the Edge Service via Gateway, which is an element of Kubernetes' Gateway API that consists of the following three elements:

- **GatewayClass:** It defines a set of Gateways with common configurations and is managed by a controller that implements the class. We will use the Gateway implemented by Nginx, NGINX Gateway Fabric.
- **Gateway:** It defines a network endpoint that can be used to process traffic on a backend such as a Service.
- **HTTPRoute:** It defines the rules and behavior for routing HTP requests from a Gateway to backend network endpoints, such as those of a Service.

We will only see the configuration of the Gateway and HTTPRoute, while the configuration of NGINX Gateway Fabric is done via the **`run-k8s-env.sh`** script that you can find in the Git repository at the path **`infra/kind`**.

We define the Gateway resource as follows:

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: gateway
spec:
  gatewayClassName: nginx
  listeners:
    - name: http
      port: 80
      protocol: HTTP
      hostname: "*.easyshop.com"
```

The manifest creates a gateway using the Nginx controller (**`spec.gatewayClassName: nginx`**). We also expose the gateway on port 80 of the hostname that matches the regular expression "*.easyshop.com".

We finally associate an HTTPRoute with the Gateway we just created, as shown below:

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: edge-service
spec:
  parentRefs:
    - name: gateway
      sectionName: http
  hostnames:
    - "api.easyshop.com"
  rules:
    - matches:
        - path:
            type: PathPrefix
            value: /
```

```
    backendRefs:
      - name: edge-service
        port: 8090
```

The HTTPRoute just created is associated with the Gateway seen above due to the **"spec.parentRefs.name: gateway"** field. The specified hostname is a subset of the one specified by the gateway. For that hostname, all paths are matched (**spec.rules.matcher.path: /**). Finally, the HTTPRoute is matched to the Service of edge service (**spec.rules.backendRefs.name: edge-service**).

You can find the file containing deployment and service resources of catalog service in the file **catalog-service/k8s/catalog-service.yml**, and those inherent to egde-service (including Gateway) in the file **edge-service/k8s/edge-service.yml**.

## Running the cluster

From the terminal, go to the path **infra/kind** and run the run-k8s.sh script with the command:

```
./run-k8s-env.sh
```

The script performs the following actions:

- Create the Kubernetes cluster using Kind.
- Imports edge service and catalog service images into nodes in the cluster to be used by Pods.
- Create deployments and services for all tools used by egde service and catalog service, such as Postgres, Keycloak, Grafana, and so on.
- Finally, deploy NGINX Gateway Fabric on the cluster.

The script code is as follows (all files used by the script are available in the Git repository of this chapter):

```
#!/bin/bash


echo 'START RUN K8S ENV'
echo 'CREATING K8S CLUSTER'
kind create cluster --config kind-multi-node-cluster.yaml --name easyshop-
cluster
kubectl cluster-info --context kind-easyshop-cluster


echo 'IMPORTING CATALOG SERVICE AND EDGE SERVICE IMAGES'
kind load docker-image catalog-service:0.0.1-SNAPSHOT --name easyshop-
cluster
```

```
kind load docker-image edge-service:0.0.1-SNAPSHOT --name easyshop-cluster

echo 'CREATING POSTGRES DEPLOY'
kubectl apply -f postgres/postgres.yml
sleep 5
echo 'CREATING KEYCLOAK DEPLOY'
kubectl apply -f keycloak/keycloak-config.yml
kubectl apply -f keycloak/keycloak.yml
echo 'CREATING REDIS DEPLOY'
kubectl apply -f redis/redis.yml
echo 'CREATING TEMPO DEPLOY'
kubectl apply -f observability/tempo/tempo.yml
echo 'CREATING LOKI DEPLOY'
kubectl apply -f observability/loki/loki.yml
echo 'CREATING PROMETHEUS DEPLOY'
kubectl apply -f observability/prometheus/prometheus.yml
echo 'CREATING GRAFANA DEPLOY'
kubectl apply -f observability/grafana/grafana-datasources-config.yml
kubectl apply -f observability/grafana/grafana-dashboards-config.yml
kubectl apply -f observability/grafana/grafana.yml

echo 'CREATING NGINX GATEWAY' # see https://docs.nginx.com/nginx-gateway-
fabric/installation/running-on-kind/
kubectl kustomize "https://github.com/nginxinc/nginx-gateway-fabric/config/
crd/gateway-api/standard?ref=v1.4.0" | kubectl apply -f -
kubectl apply -f https://raw.githubusercontent.com/nginxinc/nginx-gateway-
fabric/v1.4.0/deploy/crds.yaml
kubectl apply -f https://raw.githubusercontent.com/nginxinc/nginx-gateway-
fabric/v1.4.0/deploy/default/deploy.yaml
kubectl apply -f nginx.yaml
sleep 15
kubectl get pods -n nginx-gateway
echo 'FINISH RUN K8S ENV'
```

Once the script is launched, wait until the Keycloak Pod is in the ready state. You can see the list of running Pods and their status with the following command:

```
kubectl get po
```

You can filter the Pod list for a selector, using the following command:

```
kubectl get po -l app=easyshop-keycloak
```

Also, if you do not want to run the above command multiple times to check the Pod's state, you can run the following command that waits (until a certain timeout is specified in the input) for the Pod to be in the ready state:

```
kubectl wait --for=condition=ready pod -l app=easyshop-keycloak
--timeout=120s
```

Once the Pod of Keycloak is in the ready state, we can create the deployments and services of the catalog service and edge Service by running the following commands from the root path of the **`Chapter-09`** folder:

```
kubectl apply -f catalog-service/k8s/catalog-service.yml
kubectl apply -f edge-service/k8s/edge-service.yml
```

Running the command again to display all running Pods, you should get the output like this:

```
kubectl get po
NAME                                READY   STATUS    RESTARTS      AGE
catalog-service-77dd8df755-xmxgf    1/1     Running   0             2m8s
easyshop-grafana-6c9f7d885c-h9zhp   1/1     Running   0             12m
easyshop-keycloak-9cdcb8d96-wrqjw   1/1     Running   0             12m
easyshop-loki-7bc45f4476-nhvb2      1/1     Running   0             12m
easyshop-pgadmin-84b445977f-f5768   1/1     Running   0             12m
easyshop-postgres-7558dd565f-kbrth  1/1     Running   0             12m
easyshop-prometheus-b5b8fd7d7-8lbqr 1/1     Running   0             12m
easyshop-redis-84b88654cf-9f5rv     1/1     Running   0             12m
easyshop-tempo-8498b7945d-vjgb6     1/1     Running   0             12m
edge-service-7c9d9bf965-z895q       1/1     Running   0           2m4s
```

We open two terminals to view the Pod logs of edge service and catalog service, using the following commands:

```
kubectl logs -f -l app=edge-service
kubectl logs -f -l app=catalog-service
```

In order to access the Gateway locally, we will use port-forwarding on the nginx-gateway Pod deployed by the **`run-k8s-env.sh`** script is seen earlier. First, let us see what the name of the **`nginx-gateway`** Pod is using the command:

```
kubectl get po -n nginx-gateway
```

Since the Nginx Gateway is deployed to a custom namespace, we specified the "-n" option in the command above. Once the Pod name is displayed, we run the command:

```
kubectl -n nginx-gateway port-forward nginx-gateway-7bbb54d7-p2d4s 8080:80
```

By doing so, we will be able to invoke the Gateway locally using the endpoint **http://api.gateway.com:8080**.

Open an incognito tab in your browser and type the URL: **http://api.easyshop.com:8080/products/LAP-ABCD.** You will be redirected to the Keycloak login page. Enter the **`mrossi/mrossi`** credentials, after which you should see the JSON containing the product information with the LAP-ABCD code. From the edge service or catalog service logs, extrapolate the TraceID of the request and view the Grafana dashboards at the URL **http://localhost:3000** to verify that the observability tools are also running properly on the Kubernetes cluster.

Due to Kind, we were able to deploy Easyshop on a Kubernetes cluster. However, the Deployment and Service files created for the edge service and catalog service are also compatible with a Kubernetes cluster managed by a cloud provider.

# Conclusion

In this bonus chapter, we saw how with Spring Boot it is easy to create container images without the need to write a Dockerfile. This leads to several advantages, such as not having to maintain the Dockerfile, not having to worry about writing a secure and efficient Dockerfile, and allowing you to create reproducible images.

We first released edge service and catalog service via Docker Compose, just as we did for other tools such as Postgres and Keycloak in previous chapters. Then, we released the two microservices and all the Easyshop tools on a Kubernetes cluster, using the Kind tool, which allows us to create nodes locally. With Spring Boot, we do not need to modify the applications to release them on Kubernetes. Rather, Spring Boot Actuator provides out-of-box probe endpoints used by Kubernetes to understand whether the Pod is alive (liveness probe) and ready to receive HTTP requests (readiness probe).

We have reached the end of this book. You are now ready to release into production Spring Boot applications that can use synchronous APIs efficiently, using the most popular paradigm, REST, but not only that. You are now able to evaluate on a case-by-case basis whether it is cost-effective to use GraphQL or gRPC. In addition, you have learned about the Spring Cloud Stream project that allows you to write asynchronous APIs consistently beyond the message broker used.

# Points to remember

- Spring Boot allows the creation of container images without the need to write Dockerfiles.
- The values of properties declared in the application.properties file, are overridable via environment variables using a specific naming convention.
- Releasing a Spring Boot application on a Kubernetes cluster requires no code changes.
- Spring Boot Actuator provides out-of-box liveness and readiness probe endpoints.

# Exercises

1. Try releasing the other microservices on the Kubernetes cluster as well.
2. Experiment with scaling the replicas of the deployed services and observe how Kubernetes distributes the load.
3. Configure liveness and readiness probes for one of the microservices and test their impact on service availability during a simulated failure.

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

**https://discord.bpbonline.com**



# Index


## Symbols

@KafkaListener 181

## A

Apache Kafka 177, 178
Apache Kafka, bindings 185
Apache Kafka, components
  Broker 177
  Cluster 177
  Consumer 177
  Consumer, group 178
  Partitions 177
  Producer 177
  Topic 177
API-First Approach 19
API-First Approach, advantages 19
API Gateway 224
API Idempotence 12, 13
API Idempotence, methods 13
AsyncAPI 203-206
Asynchronous Communication 5, 171
Asynchronous Communication, points
  HTTP Polling 171, 172
  HTTP Streaming 173
  Pub/Sub 176
  Webhook 172
  WebSocket 175

## C

Circuit Breaker, states
  Closed 229
  Half-Open 229
  Open 229
Container 278-280

**D**
DELETE 13
Distributed Tracing 267
Docker 280
Docker Compose 284-287
Docker Desktop 32
Dockerfile 281

**E**
Event Loop 31
Event Loop, advantages
  cost, efficiency 31
  resource, efficiency 31
  scalability 31

**F**
ForkJoinPool 76

**G**
GET 13
Grafana 270-275
Grafana Loki 260-262
GraphQL 112
GraphQL APIs, testing 138-140
GraphQL, configuring 118-128
GraphQL, fundamentals
  Queries Mutations 114-118
  Schemas Types 113, 114
GraphQL, points
  Document Prase,
    caching 134-136
  Error, handling 132-134
  N+1 Problem 128-132
  Spring 137, 138
GraphQL/REST,
    comparing 140, 141
gRPC 144
gRPC APIs, testing 162-166
gRPC, architecture 150, 151
gRPC Client,
    implementing 157-161
gRPC Communication,
    types
  Bidirectional, streaming 148
  Client, streaming 147
  Server, streaming 147
  Unary RPC 147
gRPC, configuring 145-147
gRPC/REST, differences
  Communication,
    protocols 148
  Data Format 149
  Performance 149
gRPC/REST, use cases 150
gRPC Server,
    implementing 151-157
gRPC Server, points 153, 154

**H**
HTTPie 33
HTTP Polling 171, 172
HTTP Streaming 173, 174

**J**
Java 32
JSON Web Token (JWT) 217, 218

**K**
Keycloak 222, 223
Keycloak, components
  Clients 223
  Realms 223
  Roles 223
  Users 223
Keycloak, steps 223, 224
Kubernetes 287
Kubernetes Cluster,
    running 302-305

Kubernetes, components
  Cluster 288
  Control Plane 288
  Ingress/Gateway 288
  Pod 288
  Worker, nodes 288
Kubernetes,
    deploying 294, 295
Kubernetes, elements
  Gateway 300
  GatewayClass 300
  HTTPRoute 300
Kubernetes, features
  Cloud Ecosystems,
    integrating 288
  Dynamic, scalability 287
  Load, managing 288
  Orchestration 287
  Self, healing 287
  Updates, rolling 288
Kubernetes, points 297
Kubernetes, services
  LivenessProbe 295
  ReadinessProbe 295
  StartupProbe 295

**L**

Logging 259
Logging, field
  app_name 259
  correlation_id 260
  logger_name 260
  log_level 259
  Message 260
  process_id 259
  Separator(--) 259
  Thread_name 260
  Timestamp 259

**M**

macOS API, creating 292-294
Metrics, monitoring 262
Microservice Architecture 2, 3

**O**

OAuth2 220, 221
OAuth2 Client,
    configuring 234
OAuth2 Client Token,
    managing 243, 244
OAuth2, points
  Client Credentials 221
  Implicit 221
  Password Credentials 221
OIDC, actors
  Authorization, server 218
  Client 218
  User 218
OIDC, flow 219
OIDC, steps 219
OIDC With OAuth2,
    testing 251-253
OpenAPI 16, 17
OpenID Connect
    (OIDC) 218
OpenTelemetry 268, 269

**P**

Pagination API 90-99
Platform/Virtual Threads,
    comparing 72, 73
POST 13
ProductResponse 51
Pub/Sub 176, 177
PUT 13

**R**

RBAC 246

RBAC, role
  Resource Server,
    preventing 248-251
  Token ID 246-248
Reactive Paradigm 22
Reactive Paradigm,
    characteristics
  Elastic 23
  Message-Driven 23
  Resillient 23
  Responsive 23
Reactive Paradigm,
    concepts 22-24
Reactive Streams 24
Reactive Streams Code,
    analyzing 27
Reactive Streams, concepts 25
Reactive Streams, operations 26
Reactive Streams, operators
  publishOn() 28
  subscribeOn() 28, 29
Reactive Streams,
    responsibilities
  Processor 26
  Publisher 25
  Subcriber 26
  Subscription 26
Reactive Streams Schedulers,
    methods
  Scheduler.boundedElastic() 28
  Schedulers.immediate() 28
  Schedulers.parallel() 28
  Schedulers.single() 28
Resilience4j 230
Resilience4j, parameters 231
Resource Server, configuring 242
Resource Server,
    implementing 244, 245
REST APIs 7
REST APIs, architecture 8
REST APIs, considering 15
REST APIs, levels
  HATEOAS 12
  HTTP Verbs 10, 11
  Plain Old XML (POX) 9
  Resources 10
REST APIs, principles
  Cacheability 8
  Client-Server,
    architecture 8
  Layered System 8
  Stateless 8
  Uniform, interfaces 8
REST APIs, tips 15, 16
RestClient 99-104
Retry Filter, parameters
  Exceptions 228
  Methods 228
  Retries 228
  Series 228

## S

Sidecar Containers,
    deploying 289-291
Sidecar Containers, types
  ClusterIP 291
  ExternalName 292
  LoadBalancer 291
  NodePort 291
Single Page Application
    (SPA) 253
SPA, configuring 253-255
Spring 179
Spring Boot 281
Spring Boot 3 267
Spring Boot Actuator 263

Spring Boot Actuator,
configuring 263-267
Spring Boot,
breakdown 282, 283
Spring Boot, components
Builder 281
Buildpacks 282
Stack 282
Spring Cloud Function 182
Spring Cloud Function,
configuring 182-184
Spring Cloud Function,
points 182
Spring Cloud Gateway 225, 226
Spring Cloud Gateway,
configuring 226-228
Spring Cloud Gateway,
elements
Filter 225
Predicate 225
Route 225
Spring Cloud Gateway,
pattern
Circuit Breaker 229
Resilience4j 230
Retry Filter 228, 229
Spring Cloud Stream 181-188
Spring Cloud Stream Application,
testing 209-213
Spring Cloud Stream,
components
Bindings 184
Destination Binders 184
Message 184
Spring Cloud Stream Error,
handling 206-209
Spring Cloud Stream,
fields 186
Spring Cloud Stream,
uses 201-203
Spring Cloud Stream With Consumer,
implementing 196-201
Spring Cloud Stream With Producer,
implementing 190-195
Spring, components 179
Spring, configuring 179-181
Spring MVC 77, 104-106
Spring MVC,
annotation 78
Spring MVC,
architecture 77, 78
Spring MVC, initializing 79-90
Spring MVC/Spring WebFlux,
comparing 66-69
Spring MVC/Virtual Threads,
comparing 106-108
Spring Security,
configuring 235-238
Spring Security,
methods 237
Spring Session 241
Spring Session,
managing 241, 242
Spring Session,
modules 241
Spring WebFlux 31, 32
Spring WebFlux,
configuring 36-50
Spring WebFlux,
debugging 58-60
Spring WebFlux Domain,
designing 35, 36
Spring WebFlux Endpoints,
analyzing 60, 61
Spring WebFlux Exception,

Spring WebFlux, modules
  Data R2DBC 33
  Lombok 34
  PostgreSQL 34
  Reactive Web 33
  Testcontainer 34
  Validation 34
Spring WebFlux,
    properties 40, 41
Spring WebFlux,
    setup 33
Swagger 18
Swagger, features
  Codegen 18
  Editor 18
  UI 18
Synchronous
    Communication 4

## T

Thread-Per-Request 30, 31

## V

Versioning 14
Versioning, strategies
  Content Negotiation 14
  Headers 14
  MAJOR 15
  MINOR 15
  PATCH 15
  Query, parameters 15
  URI 14
Virtual Threads 72
Virtual Threads,
    configuring 73-76
Virtual Threads,
    simplifying 73

## W

WebClient 61-63
WebClient, testing 63-66
WebFlux/Visual Threads,
    differences 108
Webhook 172, 173
WebSocket 175, 176